CHAPTER 14

Understanding the Distributed Component Object Model

The *Distributed Component Object Model (DCOM)* is an add-on technology to COM. In its simplest form, DCOM allows a client to call on the services of a server located on a remote machine. More than that, DCOM allows the server to execute on the remote machine, which frees up resources on the client.

Unlike COM, where the resource location must be known and registered on the client machine, DCOM allows communication between objects on a LAN, WAN, or even the Internet. Yes, you must know the location of the object at some point (otherwise, the application would never find it), but DCOM relaxes some of the requirements for having the object registered locally.

This chapter helps you understand DCOM. The first section of the chapter is devoted to a theoretical discussion of what DCOM is all about and what it means to you as a programmer. We'll look at the DCOM architecture as a whole and the various methodologies you need to know to actually implement DCOM.

Once you have a better understanding of what DCOM is all about, we'll create a very basic COM component and activate it using DCOM. In other words, the component will reside on one machine and the client on another machine. The component will execute on the server, and the test application will execute on the client machine. We'll look at the implications of this distributed form of processing and some of the problems you may encounter using it in the example.

In This Chapter

Understanding how DCOM works

Working with networks to create COM connections

Understanding the types of DCOM connections

Making a component perform dual-duty using local and remote connections

Knowing the limitations of DCOM connections

Writing a DCOM client application

Writing a DCOM enabled component

Working with the DCOM configuration tool

Understanding how DCOM and monikers work together

Since this chapter talks about remote COM communication, you must have a minimum of two Windows machines set up in a network configuration to work with the following exercises. Make certain that both machines support DCOM, which leaves out older versions of Windows. All versions of Windows NT 4.0 ship with DCOM. You can download a copy of DCOM for the first version of Windows 95 from Microsoft's Web site; later versions of the product include DCOM as a default.

Microsoft also provides you with some utility programs for working with DCOM. The most important of these tools is the DCOM Configuration Tool. The next section of the chapter helps you understand how this tool works and when you can use it. Of course, you also can make use of all of the COM tools you've used in the past. Chapter 12 looks at several of these tools. The two tools that you should know how to use for working with the examples in this book are the ActiveX Control Test Container and the OLE/COM Object Viewer.

The final section of the chapter discusses DCOM and monikers. A *moniker* is simply a convenient name for a resource on another machine. This section explores monikers in general and then applies them to DCOM. You can discover how using monikers will allow you to access resources on other machines without really knowing a lot about the other machine's configuration. In other words, using monikers helps you avoid the problems of using specific registry entries to make the connection between a container and server.

An Overview of the DCOM Architecture

In this section, I answer a lot of theoretical questions about DCOM. For example, many of you may not have any idea why DCOM is such a great architectural addition in the first place. We explore the question of why you'd use DCOM. Obviously, I also answer the question of when DCOM is either overkill or inappropriate (when other technologies can take care of the situation better).

The next thing we discuss is what makes DCOM different from COM (other than the fact that the COM object executes on one machine and the application on another). It's important to have an overall understanding of what makes the technology work.

Finally, we visit the DCOM network protocol in detail. We get into the very depths of DCOM so that you can see how the various mechanisms it contains work.

Why DCOM?

There are a lot of reasons to use DCOM. Of course, the simple reason is that it's one of the existing technologies that allow you to create distributed applications. However, DCOM provides a lot more than just the ability to run COM objects from a remote location. The following list provides just a few reasons why using DCOM is essential in some situations.

Fault Tolerance	DCOM can endure both hardware and network failures. What this means is that you gain some level of reliability. Obviously, the amount of gain depends on your current network configuration and what you expect DCOM to do.
Scalability	Theoretically, you could write a single DCOM object and use that same object no matter how large your company gets. There are some practical considerations you need to take into account, such as a multiple server scenario, but for the most part, DCOM is more scalable than COM.
Load Balancing	One of the problems with the desktop right now is that one machine might have an overloaded process, whereas another wastes processor cycles waiting for work. DCOM helps you balance the load on your entire network and makes it more efficient by using more of those wasted resources.
Maintenance Ease	Placing a single COM object on the server, then allowing all of the clients on your network to access it is one way to reduce the amount of time required to implement a solution. Initial releases and updates can now be performed by making one change to a single machine.
Flexibility	DCOM hides many implementation details from the client. The client thinks that it always has a local connection to the server, which greatly simplifies the development process. Server hiding allows you to place the server where it's needed without rewriting any application code. Using DCOM also enables you to tune the application somewhat for maximum performance by creating the best possible environment for both client and server. For example, you can move the server object from one machine to another when resources on the first machine become strained. All you need to do on the client machine is make a simple configuration change in the registry.

Connection Management DCOM uses the same reference counting mechanism as COM does to know when it's safe to release a component. However, DCOM goes beyond reference counting when it comes to the connection itself. Since network connections are uncertain at best, DCOM sends a periodic "ping" to the client (at approximately two-minute intervals). If the client doesn't respond for three consecutive pings, then DCOM considers the connection broken and decrements the component count. This ensures that the component remains active only when there are active clients to service. You'll also find that DCOM allows both client/server unidirectional communication and the peer connections required for two-way communication.

As you can see, there are a lot of good reasons to use DCOM in a distributed application environment. Of course, that begs the question of what a distributed environment is and what kind of applications it can support. A distributed environment occurs whenever two or more machines are physically involved with application execution. In other words, an application that relies on data stored on a second machine doesn't constitute a distributed environment.

There are actually two kinds of distributed environments, at least in the context of this discussion. The first type involves an application designed from the ground up for a distributed environment that really doesn't work any other way. Applications that fall into this category include teleconferencing and many forms of database management. The second type consists of two machines working together toward an end result. The application can just as easily execute on a single machine — the main advantage from distribution is to gain some performance advantage through load balancing.

Obviously, you don't get all of the advantages the DCOM provides for free; there's always some cost involved with implementing any technology. These drawbacks help determine when DCOM is the appropriate choice for a specific application programming need. For example, you may not want to use DCOM in place of COM when all processing can be performed locally. The following list provides an overview of some of the problems that occur when working with DCOM.

Security Any time you send data over a network, you take the risk of allowing someone to see it. COM not only deals with code, it also works with data. Since the component is executing on the server, the only thing that gets transferred over the network connection is the data, which is also the element that you want to protect most. As with any distributed technology, DCOM presents many security issues that you need to consider before implementation.

Performance There are a lot of performance inhibitors when it comes to DCOM, so there isn't any way you can expect to get desktop performance from it unless the remote machine offers a great deal in the way of resources such as memory, hard disk space, and raw processor speed. In short, the remote machine needs to process the information at a much higher rate of speed than the client to get what appears to be desktop performance for that particular user. The first performance problem is the connection itself. Using DCOM means that you go through more layers to create and maintain a connection to the server. Of course, the network connection itself adds further delays. Finally, there's server latency to consider — the time the component waits for the server to allow it to execute.

Reliability DCOM normally relies on the *User Datagram Protocol (UDP)* subset of TCP/IP to transfer data when this protocol is available. UDP is a connectionless protocol that relies on a combination of an IP address and port number to get data from one point to another. Unfortunately, UDP can present a number of problems, including the inability to guarantee a successful data transfer. The use of acknowledgments does reduce the impact of using UDP. In addition, using UDP does allow DCOM to perform several data transfer optimizations by merging acknowledge packages with data and pinging (client keep-alive) messages.

Though DCOM does prefer to use UDP when available, there isn't any rule that says your application must use UDP. DCOM is flexible enough to allow the use of other protocols — even custom protocols. Many of the shortcomings mentioned here can be overcome using custom protocols. For example, you can overcome the security issues by using a protocol that sends the data in encrypted form.

Connection Availability Any distributed application needs to deal with the problem of lost or garbled connections. Additionally, there isn't any guarantee that a connection will be available when the user actually needs it. Although solutions like data caching do alleviate part of the problem, you still need to consider connection management when building a COM object for distributed use.

An Overview of the Connection

One of the main things you need to worry about when working with a remote access protocol such as DCOM is precisely how the connection is to take place. After all, without a good connection, many other considerations are rendered moot.

This section of the chapter answers several questions. First, we'll look at how the connection actually works — at least from an overview perspective. Second, we'll look at the question of how DCOM keeps the connection from becoming so mired in detail that the user gets tired of waiting for something to happen. Third, we'll look at some load balancing concerns — it's essential to know how DCOM scales as the load on one or more components in an application increases.

How Does the Connection Work?

In this section, we'll take a bird's eye view of how DCOM creates and manages a connection between the client and server. It's important to understand how this connection works so that you can troubleshoot problems in your own applications. Figure 14-1 contains a block diagram of the flow of data from the client to the server. The following list describes each of the diagram elements.

Client	Originates requests to the server for resources and support.
OLE32	A DLL containing the methods used to create an instance of an object (along with a wealth of other functionality). There are five methods available for remote object creation: CoCreateInstanceEx( ), CoGetInstanceFromFile( ), CoGetInstanceFromStorage( ), CoGetClassObject( ), and CoGetClassObjectFromURL( ).
Service Control Manager (SCM)	Creates the initial connection between the client and server. The SCM is used only during object creation. This feature gets implemented by the TransportLoad( ) method in the RPCLTSCM.DLL file.
Proxy	The server's presence within the client's address space. The proxy, which is actually a table of interfaces, is created and managed by the operating system at the request of the COM runtime. It allows the client to think that the server is local, even though the server is actually located on another machine. Windows uses a method such as CreateProxyFromTypeInfo( ) found in the RPCRT4.DLL file to create the proxy.
COM Runtime	Operating system elements that host objects and provide client/server communication. The COM runtime is part of any COM-related scenario, both in-process and out-of-process, local and remote.

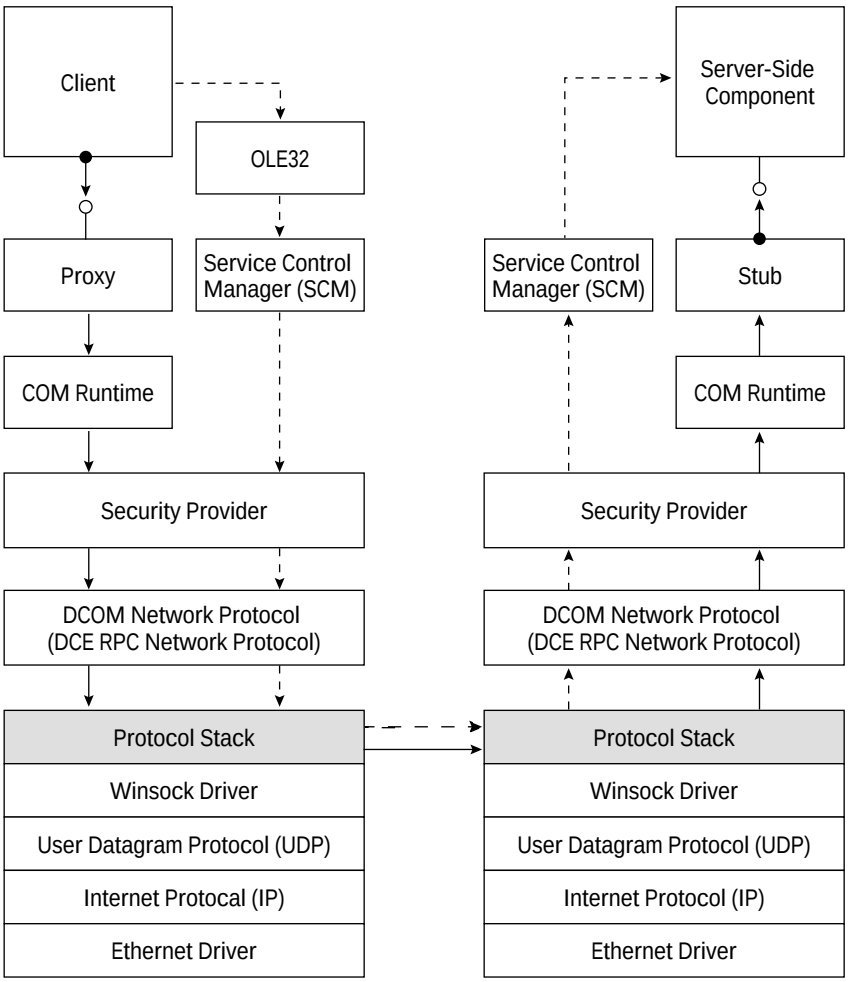

Figure 14-1: DCOM connection overview block diagram

Security Provider

The security provider logs the client machine into the server machine. Windows 2000 provides support for several standard security providers for both Internet and local network use. These providers include NT LAN Manager (NTLM, the standard Windows NT security protocol), Kerberos, Distributed Password Authentication (DPA, which is used by CompuServe and MSN), secure channel security services such as Secure Sockets Layer (SSL), Private Communication Technology (PCT), and third-party

	Distributed Computing Environment (DCE) providers. Some security providers also ensure that all data transferred between the client and server is protected in some way — usually through the use of encryption.
DCOM Network Protocol (DCE RPC Network Protocol)	Defines a protocol for creating a connection with a remote server for the purpose of using objects. Chapter 12 covers the whole issue of DCE Remote Procedure Call (RPC) with regard to component programming in the "Out-of-Process versus In-Process Servers" section of Chapter 12. In addition to implementing a component protocol, this block contains all of the elements to implement the Object Remote Procedure Call (ORPC) specification at an application level. This particular component is known by several different names in the Microsoft documentation — the most popular of which is DCOM wire protocol. We'll discuss the DCOM network protocol in detail later in this chapter.
Protocol Stack	Actual network communication requires more than just one protocol — there are network-related protocols to consider as well. The protocol stack consists of all the protocols required to create a connection between the client and server, including network-specific protocols such as TCP/IP. Figure 14-1 shows a typical protocol stack consisting of a Winsock driver, UDP, IP, and an Ethernet driver. Not shown is the Ethernet network interface card (NIC) actually used to create the physical connection between the client and server.
Stub	The client's presence within the server's address space. The stub is created and managed by the operating system at the request of the COM runtime. As far as the server is concerned, the stub works with a local client. Windows uses a method like CreateStubFromTypeInfo( ) found in the RPCRT4.DLL file to create the stub.
Server	The COM object that the clienthas requested services and resources from.

Figure 14-1 actually shows two communication paths. The first path (dotted line) is used to create an instance of the object. The second path (solid line) is used for normal communication between the client and the server.

Creating a line of communication between a client and server normally follows these steps.

1. Client issues one of the five object creation method calls, as discussed earlier in the section, to OLE32.DLL. The call must include both a class ID (CLSID) and a server name (along with any information required to log onto the server). As an alternative, the client can issue a standard call that OLE32.DLL resolves to a remote location based on a registry entry, or the client can use monikers.
2. OLE32.DLL calls upon the client-side SCM to create a connection to the server machine since it can't service the call locally.
3. The DCOM network protocol creates the required packets to send information from the client to the server.
4. The server-side SCM creates an instance of the desired server-side component and returns a pointer of the object instance to the client.
5. The server-side SCM calls upon the COM runtime to create a stub for the component to interact with.
6. The client-side SCM calls upon the COM runtime to create a proxy for the client to interact with.
7. The SCM returns a pointer to the proxy to the client.
8. Normal client- and server-side component communications begin.

Connection-oriented Data Flow Optimization

Sometimes, the development and placement of a component depends on just how much communication is taking place. Consider a situation where a client needs access to the contents of a database. You can place a component directly on the client machine that accesses the database manager, gains access to the required data, then formats it for the user. However, this design requires a lot of network communication because the client needs to constantly communicate with the database manager and send or receive the data.

Splitting the component in two would allow you to reduce the amount of data traversing the network. One component on the client machine can send data requests and format the incoming data. A second component on the server can make the data requests and deliver only the required information to the client. Using this approach significantly reduces network traffic, enhances both client and server efficiency, and makes the user more productive all at the same time.

DCOM does perform some connection manipulation on its own. One of the most important changes that DCOM implements automatically is connection optimization. For example, if you have a server-side component that's manipulating a database using ODBC, DCOM more than likely will copy the component to the client, then get out of the picture. Since the connection to the database is through ODBC, neither the client, nor the database manager, notice any difference in the performance of the component. However, since DCOM is out of the picture, the component executes more efficiently. Obviously, this is a very specific kind of connection change and is implemented only when the client sees a significant performance gain.

Load-balancing Concerns

Load balancing is another connection-related problem. Normally, you begin with all of the server-side components required for an application loaded on a single machine. As people get added to the application user's list, you may have to move some components to their own machine to avoid overloading a single machine. Breaking the component into smaller pieces may relieve some server stress as well. However, at some point, even a well-designed component may overload a single server. At this point, you need to balance the component load across multiple servers.

DCOM intrinsically supports two types of load balancing — static and dynamic. The first relies on the assignment of users to specific servers, and the second relies on component referral techniques. Windows 2000 adds a third form of load balancing in the form of a COM+ feature called, oddly enough, Load Balancing. This third form of load balancing relies on one or more routers and a cluster of servers.

Static load balancing occurs when you assign specific users to specific servers. Older versions of Windows normally rely on registry entries to enforce the assignment of user machines to a specific server. Fortunately, administrators can use the Win32 remote registry functions to make any changes.

Another method to statically load balance DCOM connections is to use a database containing user names and the servers they're assigned to. Although this method does allow the administrator to make reassignments quickly, it also means adding some processing overhead to the connection and complexity to the application. The overhead comes into play in several ways — the most important of which is the time required for the client to look up the appropriate server prior to requesting component services. In addition, the client has to be rewritten to use the database, which may make the application unnecessarily complex and inflexible.

Yet another method of overcoming problems with static load balancing is to use a referral component. In this case, the client is designed to request the component's services from the referral component at the outset, so no extra programming is required. Once the referral component receives a request, it automatically makes the connection for the client, reducing the amount of overhead from using this technique. The obvious downside of this form of static load balancing is that you need to write a special referral component.

Active Directory is the storage medium of choice for Windows 2000 developers. Instead of placing the DCOM entry in the registry, it appears in the Active Directory centralized storage. The actual storage location is called the COM+ Catalog; it unites the COM and Microsoft Transaction Server (MTS) registry entry methods. Obviously, this second method allows the administrator to make all required configuration changes from one location to a store that's always online, rather than to multiple registries, some of which may be offline because the associated machine is turned off. In addition, this second method allows the administrator to assign a specific user to a specific server rather than assigning a machine to the server.

Both developers and administrators will be able to interact with the COM+ Catalog using the COM+ Explorer. In addition, Microsoft is developing a series of new COM interfaces that will allow the developer to interface with the COM+ Catalog directly.

Two of the major problems with static load balancing are that you need a predictable load and administrator intervention to implement it. In other words, static load balancing probably works fine for an order entry system where the same number of users log in each day. It probably doesn't work well for an Internet site where the number of users may vary by a large amount over the period of a few hours.

Dynamic load balancing also can rely on the referral component technique described earlier. However, in this case, the referral component takes factors such as current server load, past user request history, and current network topology into account when assigning a user to a machine rather than assigning the user based on a static list. Using this technique means that the referral component requires more time to get the job done but that the network processes requests more efficiently once the connection is made.

There are several problems with this approach. For one thing, the complexity of the referral component is greatly increased. Instead of just directing the user's request to another component, the referral component now has to make some decision as to which server to select. In addition, this technique relies on a database for storing user request history, the availability of server statistics (which must be refreshed for each request), and the availability of network topology data. In other words, dynamic load balancing is much harder to implement because you essentially replace the network administrator with some software.

Balancing a load at connection time doesn't necessarily mean that the load remains balanced either. True dynamic load balancing needs to incorporate some type of reconnect strategy. As the load from a user on one machine increases and the load from a user on another machine decreases, it may be necessary to reconnect them to different servers to keep the load on both servers balanced. Unfortunately, DCOM doesn't provide any built-in functionality to implement a reconnection strategy, which means the developer needs to create yet another component that monitors the current situation and makes reconnection recommendations as necessary.

Implementing a reconnection strategy is complicated by the requirement of the component to retain state information for the client. The component and client both need to decide that the current state information is no longer required, then request a reconnection before the next method invocation. As you can see, building a reconnection strategy is extremely complex, making the prospect of adding yet another server to the network quite appealing. The complexity of the problem is most likely one of the reasons that Microsoft didn't implement a reconnection strategy as part of DCOM itself.

Load Balancing is a new COM+ feature that's built into Windows 2000. This method assumes that you have a lot of users to manage and that the application you create adheres to Microsoft's new Distributed interNet Applications (DNA) Architecture.

Implementing the COM+ form of load balancing is relatively easy. The first thing you need is an application cluster — which is a set of up to eight machines that are capable of running the requested component. Once you have the application cluster machines loaded with the server-side components that you want to use, you need to add a load-balancing router to the picture. The load-balancing router sends client requests to one of the machines in the application cluster.

The client application won't access the application cluster machines directly. What happens instead is that client requests get sent to the load-balancing router. The load-balancing router uses an algorithm to determine which application cluster machine is least busy, then routes the request to that application cluster machine. Of course, the question that the load-balancing router needs to answer is what criteria to use for determining application cluster machine load. The default algorithm uses a response time algorithm to determine which machine to use. You can, however, write your own load-balancing engine that determines which application cluster machine is least busy. For example, you may conclude that the number of available resources is a better way to determine which application cluster machine has the lightest load if your component requires a lot of resources to execute.

The bottom line from a developer's perspective is that you write your application to direct all server-side component requests to the load-balancing router. If the configuration of the network changes, it doesn't matter because the load-balancing router still accepts all component requests. As you can see, using the COM+ methodology for balancing the component load on several servers is the easiest to implement, as long as you have the required hardware and all of that hardware is running Windows 2000.

The current MTS method for creating components in a load-balanced environment is to create the reference early and hold on to it as long as possible. This approach ensures that the client gets the resources required to complete execution of a given task and that MTS operates as efficiently as possible. The new approach for COM+ is to create components only when necessary and to release them as early

as possible to free resources. Obviously, these two approaches are completely opposite of each other, which means you may have to rewrite some of your older applications to take advantage of the latest COM+ technology. However, unless performance isn't a problem on your network, it's usually well worth the effort to rewrite the component to take advantage of this new technology.

A Detailed Look at the DCOM Network Protocol

The first thing we need to talk about in this section is what the term *DCOM network protocol* means. The DCOM network protocol is a superset of the Open Systems Foundation (OSF) Distributed Computing Environment (DCE) Remote Procedure Call (RPC) network protocol. Essentially, the DCOM network protocol defines the methodology for allowing COM to work across two machines and making it appear to both client and server that there's only one machine involved. By this point, you should understand that DCOM is simply COM that works across two machines and that there has to be a network infrastructure with required protocols involved to enable that communication.

Once you do a little reading, you'll find that DCE is actually composed of several components, only one of which is RPC. There are also the Cell and Global Directory Services (CDS and GDS), the Security Service, DCE Threads, Distributed Time Service (DTS), and Distributed File Service (DFS) components. In addition, DCE isn't a standalone protocol; it's designed to allow an operating system vendor to include its services within an operating system. This is precisely what Microsoft has done with DCOM—by basing DCOM on DCE RPC.

The following section helps you better understand the DCOM network protocol and where it fits in the scheme of things for you as a programmer. I won't cover every last detail of this protocol since there are entire volumes on the topic. Instead, I'll cover the highlights of DCOM network protocol as a whole and what makes it different from the protocol that it's based on, DCE RPC.

Many Names, Same Functionality

Trying to nail down what the term *DCOM network protocol* means can prove frustrating unless you know where to look. Unfortunately, though there are a lot of Microsoft-generated drawings and presentations out there with DCOM network protocol written on them, searching the various resources that Microsoft provides sheds very little light on this technology. What you really need to look for is the DCOM wire protocol.

The DCOM wire protocol standard defines the kinds of packets that get put on the wire between the client and the server. (Obviously, there are other network protocols required to actually create the connection between client and server, as shown earlier in Figure 14-1.) The DCOM wire protocol standard defines the format of these packets and what the client and server should expect to see in them. In other words, the DCOM wire protocol specifies how the information gets physically transferred from one machine to the other.

DCOM network protocol goes by yet another name, object RPC (ORPC). The ORPC moniker shows the close relationship between DCOM and DCE RPC. This particular name for the single technology that we've been talking about throughout this chapter is the most common one. The "object" part of ORPC points to the fact that Microsoft has extended RPC to support remote handling of objects. In fact, Microsoft has added several additional data structures — the most important of which is the interface pointer identifier (IPID) — discussed later in the chapter. These additional data structures allow DCOM to request remote instantiation of an object from a server.

The whole point of this particular part of the discussion is that you need to do a little digging if you really want to find out every detail about DCOM. For the remainder of this section, I'll refer to the whole array of DCOM communication names and specifications as DCOM network protocol. However, it's also important to know that there are other names you'll find this technology listed under.

Microsoft has prepared a complete specification for the DCOM wire protocol. Reading this specification gives you a better idea of precisely how DCOM transmissions are supposed to work. You can find the specification at `http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt`. One of the best places to find out more about DCE RPC is at `http://www.osf.org/public/tech/dce/info/docptrs.htm`. This site contains an overview of DCE, a copy of the DCE specification, and a wealth of other information you may find useful when learning about how DCOM works. If you have questions about DCE, access this newsgroup for help: comp.soft-sys.dce. There's also an interesting presentation of how DCOM works within the DCE environment. You can find it at `http://www.opengroup.org/dce/ proggrp/meetings/july98/FrankHayes/`.

The IRemoteActivation Interface

Like many other object technologies, DCOM network protocol relies on an interface to encapsulate the methods required to create, maintain, and eventually sever a connection between client and server. In this case, I'm not talking about a COM interface, but an RPC interface named IRemoteActivation. The IRemoteActivation interface is exposed by the Service Control Manager (SCM) on each machine (see Figure 14-1). The SCM is physically located in the RPCSS.EXE file and isn't associated with the SCM that Windows 2000 uses to manage services.

Interestingly enough, IRemoteActivation contains only one method, RemoteActivation( ), which activates the COM object on a remote machine. RemoteActivation( ) makes up for a lack of a pure DCE RPC implementation by allowing remote activation of the server by the client. In a pure DCE RPC implementation, you must run the server first, then enable the client to access it remotely. It's instructional to look at the arguments that the client passes to the server using the RemoteActivation( ) method and what the server passes back to the client. Table 14-1 shows the input arguments, whereas Table 14-2 shows the output arguments for RemoteActivation( ).

Table 14-1
RemoteActivation( ) Input Arguments

Argument	*Description*
HRpc	RPC binding handle used to make the request
ORPCthis	A data structure that contains the version of DCOM that the client uses, one or more flags that indicate the presence of additional data, the casualty ID (CID) of the client, and an array of special extensions. The client's major version number is compared to the server's major version number. If the major version numbers don't match, the server sends back an error message result rather than the desired object. A server can have a higher minor version number than the client. The CID is actually a globally unique identifier (GUID) used to link the method calls between two or more machines. For example, if machine A requests that machine B activate a component, and machine B has to request that machine C activate a component first, then the calls are casually related and DCOM generates a CID for them. The CID remains in effect until the original call machine A created is satisfied by a return call from machine B. Currently, there are only two special extensions in use in the ORPCthis structure. The first is for error information, whereas the second is for debugging purposes. I don't discuss these special extensions because of their special-purpose nature.
Clsid	Class ID of the object that the client wants to create and activate on the server. This argument is typically used if the application calls CoCreateInstanceEx( ), CoGetClassObject( ), or CoGetClassObjectFromURL( ).
pwszObjectName	A pointer to a wide character string containing path and executable file name information for the object that the client wants to create. This argument is typically used if the application calls CoGetInstanceFromFile( ).
pObjectStorage	A client-side interface pointer to an object that supports the IStorage interface. The server uses this interface to determine which object to instantiate. This argument is typically used if the application calls CoGetInstanceFromStorage( ).
ClientImpLevel	Constant value taken from RPC_C_IMP that determines the client's default impersonation level.
Mode	This argument is normally set to MODE_GET_CLASS_OBJECT when instantiating a new object. The Mode argument gets passed to the server's IPersistFile::Load( ) method if either the pwszObjectName or pObjectStorage argument contains a value.

Continued

Table 14-1 (*continued*)

Argument	*Description*
Interfaces	Contains the number of interfaces that the client requests
pIIDs	A list of interface identifiers (IIDs) that the client wants. One of the more common IIDs is IID_Unknown for the IUnknown interface.
cRequestedProtseqs	Number of protocol sequences specified in the RequestedProtseqs argument
RequestedProtseqs	An array containing a list of the protocol sequences that the client wants OXID binding handles for

Not every argument in either Table 14-1 or 14-2 is required. A RemoteActivation( ) call typically includes only the arguments required to satisfy the object creation request. For example, if you include the Clsid argument, then you normally don't need to include the pwszObjectName argument as well.

Table 14-2
RemoteActivation( ) Output Arguments

Argument	*Description*
OPRCthat	This parameter appears at the head of every protocol data unit (PDU) for COM methods. It contains a list of flags that defines the presence of other data and an extension array used for special purposes.
pOxid	The object export identifier (OXID) for the object that the server has just created for the client. This is the binding information that the client needs to connect to the interface specified by the interface pointer identifier (IPID).
ppsaOxidBindings	Contains the endpoint and security bindings required to reach the OXID. In most cases, the SCM is the endpoint, which resides at a well-known port address for each of the major protocols. For example, the SCM port for UPD and TCP is 1066. The endpoint is a UNC-based string instead of a port number when named pipes are used.
pipidRemUnknown	The IPID of the OXID's IRemUnknown (remote unknown) interface. This is the equivalent of the COM IUnknown interface, but for a remote, rather than a local, machine.

Argument	*Description*
pAuthnHint	Constant value taken from the RPC_C_AUTHN list. It provides a hint to the client about the minimal level of authentication that the server can accept.
pServerVersion	The version of DCOM supported by the server. The client and server must support the same major version number of DCOM. In addition, the server's minor revision level must be equal to or greater than that of the client, or an error will occur. If the server's minor version number is greater than that of the client, then the server automatically scales its level of DCOM support to match that of the client.
phr	The HRESULT of the activation operation. This is the COM object's instantiation return value to the client; don't confuse it with the value returned by the server itself for the operation as a whole.
ppInterfaceData	An array of interface pointers. This array contains one entry for each interface requested using the pIIDs argument and in the same order. Each interface pointer is actually a data structure consisting of two elements. The first element contains the size of an object reference data structure. The second element is the object reference data structure itself. (This data structure is discussed in the following section.) There are three kinds of object reference data structures: standard, handler, and custom. All three of these structure types begin with a signature, flags, and a GUID for the interface pointer. The end of the structure is object reference data structure type specific.
pResults	An array of HRESULTs for each <Object>::QueryInterface( ) method call performed on the server while gaining access to each of the IIDs requested by the client. This array doesn't include an object instantiation HRESULT value since that HRESULT is returned in the phr argument.

As you can see from Tables 14-1 and 14-2, the interaction between the client and server is very straightforward, and there isn't much more information required than what you provide for a local object instantiation. The big difference is how the SCM handles the call. As a matter of fact, you never need to worry about creating a RemoteActivation( ) method call unless you begin to work with DCOM directly. If you issue a standard CoCreateInstance( ) call and OLE32 finds that the object doesn't exist on the local machine, it creates the required DCOM call for you based on the information provided in the CoCreateInstance( ) call and on any registry entries it finds.

The DCOM network protocol transmits method arguments using the *network data representation (NDR)* format. The NDR format determines how the various values get marshaled into data packets for network transmission. Knowing how the DCOM network protocol formats data allows you to do a few things that you can't normally do, such as use a network sniffer to see how packets get transmitted from one machine to the next. There's one extension to NDR for DCOM so that NDR can support interface pointers. The MInterfacePointer data type (the object reference data structure mentioned in Table 14-2) doesn't contain a pointer to a vtable containing functions; it's merely an object reference. One of the more interesting things about the MInterfacePointer structure is that it begins with a signature field that spells MEOW when you look at the data stream in ASCII. Some programmers speculate that this is actually an acronym for Microsoft Extended Object Wire, but Microsoft hasn't documented it as being so. One thing is certain: You can always tell when you're looking at an object reference in a network sniffer if you see the word *MEOW* at the beginning.

Object References

From the preceding sections, you now know that the main purpose of all these data transfers between the client and server is to provide the client with access to a specific instance of an object interface using the object reference. Each interface requires a separate object reference that the client uses to access the methods that the interface contains. Object references are passed back to the client using the ppInterfaceData array (MInterfacePointer data type).

There are two main types of object reference data structures (OBJREF_STANDARD and OBJREF_HANDLER), and one ancillary type (OBJREF_CUSTOM). Each of these data structures includes a signature (MEOW), a flag specifying the kind of object reference, and a GUID containing the interface identifier. Each of the object reference data structure types also contains specialized data. The data contained in the main object reference types can be broken down as follows:

Standard Object Reference	A data structure containing a reference to the object created on the server.
Dual String Array	A data structure containing two strings. The first string contains the information the client needs to bind to the object on the server. The second string contains security information the client needs to access the object on the server.
Handler Class ID	This is only applicable to the OBJREF_HANDLER object reference data structure. It contains the GUID for the handler code.

Let's look at the standard object reference in more detail. This is a data structure consisting of five elements: flags, the number of reference counts associated with the object, the OXID for the server that owns the object, the object identifier (OID) of the object that implements the interface, and the IPID. The flags are used to define special settings for the object reference. The only flag that you ever need to worry about when working with DCOM directly is SORF_NOPING, which tells DCOM that the object interface doesn't need to be pinged to keep it alive. The cPublicRefs field, which contains the number of reference counts associated with the object, reduces network traffic by allowing a single call to replace multiple IUnknown::AddRef( ) calls.

The Network Name Difference

It's important to understand all of the theory behind how DCOM works, but at some point, you have to start working on the practical aspects of implementing DCOM. Obviously, you can create a low-level connection that manipulates DCOM itself, but for the most part, you may never need to do so. In fact, creating a DCOM connection can be as easy as creating an alternative type of registry entry (which requires no changes to your code) or adding the name of a network resource as part of your object instantiation call.

There are a variety of ways to create a DCOM connection. The most common method is to issue a special call from within your application. All five of the following function calls include a method for requesting the name of a network resource where the object is instantiated. (Notice that CoCreateInstance( ) isn't included in this list since it lacks the means to define a precise server name.)

- CoCreateInstanceEx( )
- CoGetInstanceFromFile( )
- CoGetInstanceFromStorage( )
- CoGetClassObject( )
- CoGetClassObjectFromURL( )

The method argument that you're interested in for the first four calls is pServerInfo, which is a data structure of type COSERVERINFO. This data structure contains several elements that describe the connection to the server as shown in Table 14-3. The CoGetClassObjectFromURL( ) method relies on the szCodeURL argument to point to the server containing the object that you want to instantiate.

Table 14-3
COSERVERINFO Data Structure Description

Parameter	*Description*
pwszName	A pointer to a string containing the name of the server in wide character format.
pAuthInfo	COAUTHINFO data structure pointer containing access information for the server. This data structure has to contain the information required to access the server, such as the user's name and password. Setting this argument to NULL forces DCOM to use the default security provided by the user's account.

What happens if you don't want to create special code to create a connection? DCOM also allows you to use a standard CoCreateInstance call by creating a special registry entry. Using this technique allows you to create a client application that works on any machine; it doesn't matter whether the component is local or remote.

Creating the required HKEY_CLASSES_ROOT registry entries for remote activation of a component is a two-step process, as shown here.

1. Create an HKEY_CLASSES_ROOT\APPID\<Application Identifier GUID> entry for the remote server like this:

   ```
   "RemoteServerName"="<DNS Name>"
   ```

2. Create an HKEY_CLASSES_ROOT\ClsID\<Class Identifier GUID> entry for the component like this:

   ```
   "AppID"="<AppID GUID>"
   ```

The application identifier (AppID) concept originally was introduced for security reasons. All of the COM objects that share the same AppID also get the same security levels when accessing the server. However, the AppID also allows you to avoid redundant registry entries by allowing all of the components that reside on a single server to share one AppID. This makes the administrator's job easier when a group of components move from one server to another.

Fortunately, you don't have to worry about modifying the registry entries directly (unless you want to do it as part of the application installation process). Microsoft provides the DCOM Configuration Tool, which makes the required changes for you once you provide the correct level of information. We'll look at this tool later in the chapter. All you need to know for now is that the DCOM Configuration Tool produces the same results as if you had modified the registry directly as shown earlier in this section.

At this point, you might think that modifying the user's local registry entries causes problems for a large organization even if Microsoft provides a tool to perform the task for you. The answer to this need is Active Directory. Windows 2000 provides you with the capability of creating registry entries in a central location that the user

downloads as part of the process of logging into the domain. What this means is that you create the required DCOM entries on the server and everything else happens automatically. Changes require a single change on the server as well, which should greatly ease the administrator's burden. Obviously, you must run Active Directory to gain this ease of usage benefit.

Creating a DCOM Client Application

Let's create a very simple application for this example. In fact, the only thing it will do is add two numbers together and display the result. The whole point of this application is to show you remote connection between a component on one machine and a client on another machine. This example uses the registry method of accessing the remote component (the second method is to program this information directly into the application itself).

This application is set up to work on my server. You need to substitute some of the information in the source code for your setup. Make absolutely certain that you change the pServerInfo argument in the CoCreateInstance( ) function to match the name of your server, along with any required security information.

You need to complete the application, then create the component in the next section of the chapter before you can test the result. The next section includes a "Testing the Result" section that you can use to test the DCOM connection. Make sure you follow the directions completely or you might not get the desired results. For one thing, we won't register the component on the client machine to ensure that it resides only on the server (thus ensuring that we're using a DCOM connection rather than a local connection).

CD-ROM Location: Chapt14\DCOMTest
Program Name: DCOMTest.EXESource Code Modules in Text:
DCOMTestDlg.CPP, DCOMTestDlg.h, DCOMTest.RC

Creating the Program Shell

Let's begin by creating the application shell. The following procedure gets you started. (It assumes that you already completed the basic examples in Chapter 3 by using an abbreviated procedure.)

Try This

1. Open Visual C++ if you haven't done so already.
2. Use the File | New command to display a New dialog box. Click the Projects tab.
3. Highlight the MFC AppWizard (exe) icon.

4. Type a name for your program in the Project Name field. The sample program uses the name DCOMTest.
5. Click OK. You'll see the MFC AppWizard step 1 dialog box.
6. Choose the Dialog based option, then click Next. You'll see the MFC AppWizard step 2 of 4 dialog box.
7. Change the dialog title bar text to "DCOM Component Access Example."
8. Check the Windows Sockets option. You must include this feature so that the application can communicate with the server using TCP/IP and make the required remote communication. This is about the only major difference (besides coding if you don't use the registry method) between a standard application and one that uses DCOM.
9. Click Finish.You'll see the New Project Information dialog box shown in Figure 14-2.
10. Click OK. Visual C++ generates your application. You'll see a Dialog Editing Window.

Figure 14-2: The New Project Information dialog box shows that we're creating a dialog-based application with standard features.

Designing the Dialog

At this point, you should have an application shell that includes the proper support for DCOM communications using TCP/IP. Now it's time to design the dialog box that the user sees when using the remote component. Figure 14-3 shows the simple design we'll use in this case. Notice that there are only three edit controls (two for

input and one for results), three static text boxes, and a pushbutton. Table 14-4 tells you how to configure the various controls for use.

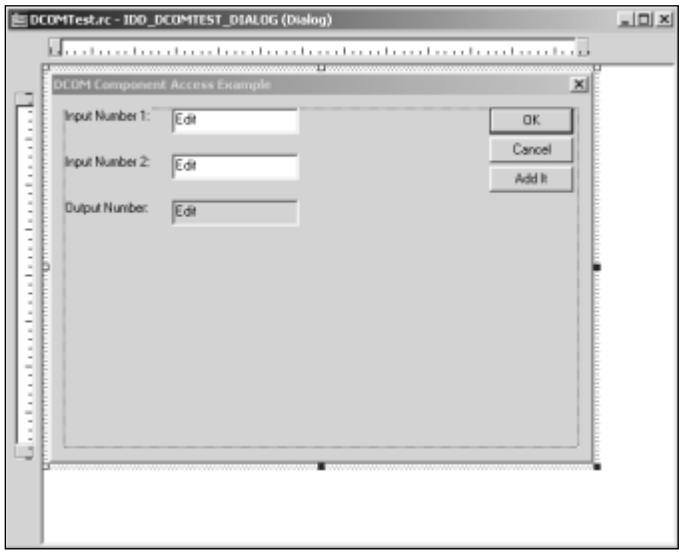

Figure 14-3: The DCOM Test dialog box is relatively simple so you can concentrate more on the process than the application.

Table 14-4
DCOM Test Component Settings

Component	*Property*	*Setting*
IDC_ADD_IT	Caption	Add It
IDC_INPUT1	Number	Checked
IDC_INPUT2	Number	Checked
IDC_OUTPUT	Read-Only	Checked

Now that you have a dialog box with some controls in it, we need to make those controls accessible to the rest of the application. The following procedure shows you how to create the member variables required for this example.

Try This

1. Ctrl+Double-click IDC_INPUT1. You'll see the Add Member Variable dialog box shown in Figure 14-4.

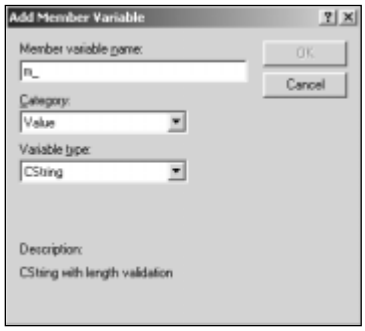

Figure 14-4: The Add Member Variable dialog box allows you to add member variables to access the dialog box components.

2. Type **m_input1** in the Member variable name field.
3. Choose Control in the Category field. Visual C++ automatically chooses CEdit in the Variable type field for you.
4. Click OK. Visual C++ adds the new member variable to your application.
5. Repeat Steps 1 through 4 for IDC_INPUT2 and IDC_OUTPUT. In this case, use m_input2 and m_output as the member variable names.
6. Use the View | Class Wizard command to display the MFC ClassWizard dialog box. Select the Member Variables tab and choose CDCOMTestDlg in the Class name field. Your list of member variables should match the ones shown in Figure 14-5.
7. Close the MFC ClassWizard dialog box.

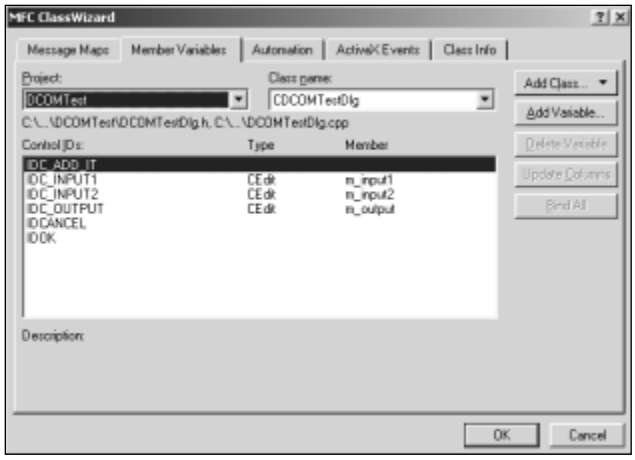

Figure 14-5: The example program requires member variables for each of the edit box components.

Adding Some Code

Our simple application is ready for some code. The first thing you need to do is add a member function to respond to the Add It pushbutton. Double click the Add It pushbutton to see an Add Member Function dialog box like the one shown in Figure 14-6. Click OK to accept the default function name. Listing 14-1 shows the code you need to add to the `OnAddIt()` function to make this application operational. (But you can't use the application until you complete the next two sections of the chapter, since the component isn't installed on the server yet and we need to configure the client to use the remote component.)

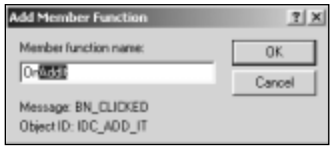

Figure 14-6: The Add Member Function dialog box allows you to add a handler for the Add It pushbutton.

Listing 14-1: **OnAddIt() Source Code**

```
  void CDCOMTestDlg::OnAddIt()
{
    int             iValue = 0;     // Result value.
    int             iInput1;        // First user input.
    int             iInput2;        // Second user input.
    CString         oString;        // Output string.
    HRESULT         hr;             // Object operation result.
    IAddNumbers     *m_pAddNumbers; // Interface pointer.

    // Convert the first input value to an integer.
    m_input1.GetWindowText(oString);
    iInput1 = atoi(oString);

    // Convert the second input value to an integer.
    m_input2.GetWindowText(oString);
    iInput2 = atoi(oString);

    // Initialize the interface pointer.
    m_pAddNumbers = NULL;

    // Initialize the COM environment.
    CoInitialize(NULL);

    // Create an instance of the object.
    hr = CoCreateInstance(CLSID_AddNumbers,
                          NULL,
                          CLSCTX_ALL,
```

```
                        IID_IAddNumbers,
                        (void**)&m_pAddNumbers);

    // If the creation failed, exit.
    if (FAILED(hr))
    {
        // Make sure the class was registered.
        if (hr == REGDB_E_CLASSNOTREG)
            AfxMessageBox("Class Not Registered");

        // Make sure we can aggregate the class.
        if (hr == CLASS_E_NOAGGREGATION)
            AfxMessageBox("Class Can't be Aggregated");

        // Display the general failure message and exit.
        AfxMessageBox("Object Creation Failed");
        return;
    }

    // Perform the math calculation.
    hr = m_pAddNumbers->DoAdd(iInput1, iInput2, &iValue);
    if (FAILED(hr))
        AfxMessageBox("Addition Failed");

    // Convert the result to a string.
    itoa(iValue, oString.GetBuffer(20), 10);
    oString.ReleaseBuffer(-1);

    // Display the output value.
    m_output.SetWindowText(oString);

    // Release and destroy the object.
    m_pAddNumbers->Release();
    m_pAddNumbers = NULL;
}
```

There isn't anything too mysterious about this code if you've worked with COM before. The first thing that the code does is initialize variables. `iInput1` and `iInput2` holds the two input variables for the addition. The `m_pAddNumbers` pointer will eventually hold the interface pointer to the AddNumbers object in the AddIt.EXE file (we'll create these items as part of the component creation process).

Once the variables are initialized, use `CoInitialize()` to turn on COM support. This is actually the first part of the object creation process. You must call either this function or `CoInitializeEx()` before you attempt to do anything at all with COM. The second part of the object creation process is to use `CoCreateInstance()` to obtain an interface pointer to the AddNumbers object. Here's a list of the parameters used to create the object.

CLSID_AddNumbers	The class identifier for the object that we want to create. This is actually a 128-bit number that Windows looks up in the registry to find out what server to use as well as other object specific information.
NULL	Contains the IUnknown interface pointer of a parent object if we want to create an aggregation (collection) of two or more components. The parent object acts as a container to hold the other objects in the aggregation. You need to see this parameter to NULL if you don't want to create an aggregation (which is why I set it to NULL for the example).
CLSCTX_ALL	Tells Windows in what context you want to run the object. I normally set this to CLSCTX_ALL just to make sure I can access the component wherever it might be at the moment. However, if you want to be absolutely certain that Windows uses the component located on a remote server, you need to set this value to CLSCTX_REMOTE_SERVER.
IID_IAddNumbers	A reference to the interface that you want to access. A class can have any number of interfaces (as we saw when creating a component in Chapter 13). This entry requests a specific interface. `CoCreateInstance()` only allows you to request one interface at a time — `CoCreateInstanceEx()` allows you to request multiple interfaces using a single call.
(void**)&m_pAddNumbers)	Holds a pointer to the requested interface on return if the call is successful. Use this pointer to access the various methods that the interface contains. This pointer is for a specific instance of the object.

It's always a good idea to perform some error trapping, especially when working with DCOM, since you can't be sure you'll have an instance of an object to work with when CoCreateInstance( ) returns. You can trap for any number of specific errors; the example shows you how to trap the two most common errors. Obviously, the class registration error is a very real possibility when working with DCOM, especially during component testing, so I make sure I include this special case test as a minimum.

Now that we have an object to work with, we can add the two numbers, which is precisely what the `m_pAddNumbers->DoAdd()` call does. Notice that the two input numbers are passed directly, but that the output variable is passed as a pointer. The reason for this is that you want the `DoAdd()` method to pass the result back into the client's address space for display. As with the `CoCreateInstance()` call, you check for error conditions after completing the `DoAdd()` call. The reason is simple: You can't rely on a remote connection to stay intact, even after you create the remote object. Once the `DoAdd()` call returns and the result is verified as good, the sample code displays the result onscreen.

At this point, you're done with the object, so you can get rid of it. Getting rid of the object is a two-step process. First, you need to call the object's `Release()` method so that the object's reference count goes to zero. Then, you set the object pointer to NULL so that any local memory allocated for it is freed.

There are two more entries you need to make to the application program. The first provides access to the component's method names. Add it to the beginning of the DCOMTestDlg.H file like this:

```
#include "..\AddIt\AddIt.h"
```

The second entry provides information about the component's interface. You add it to the beginning of the DCOMTestDlg.CPP file like this:

```
#include "..\AddIt\AddIt_i.c"
```

Creating a DCOM-enabled Component

Before we can test the application we've just created in the previous section, we need to create a component that the application can access. This is a very simple ATL-based component that doesn't provide any form of user interface. All it does is accept two numbers as input and output the sum as a result. The whole purpose of this example is to show that components used in a remote application don't require much in the way of special programming. Obviously, you do need to avoid providing a user interface since there isn't anyone at the server to interact with it.

CD-ROM Location: Chapt14\AddIt
Program Name: AddIt.DLL
Source Code Modules in Text: AddIt.CPP

This component is unique in another way. It doesn't provide a user interface, and it's not designed for including directly within the application. As a result, don't worry about creating a property page or the other paraphernalia that normally goes with working with a component in an IDE. The AddIt component is designed to interact with the client during runtime, not during design time. There are actually a lot of applications for a component like this. Any time you want the client to

request information from the server without having to ask the user for input, you can use this method. In fact, there are several advantages to using this method as listed here.

Reduced Complexity	Any time you can reduce the complexity of a component, you can also reduce the number of things that can go wrong with it, enhancing reliability as result.
Reduced Code Size	A component that doesn't include user interface elements or IDE elements will be much smaller than a component that relies on these features. Obviously, a component of this nature won't be nearly as friendly to use either. You have to weigh the benefits of reduced memory requirements and enhanced execution speed against programmer concerns.
Reduced Development Time	Graphic elements take time to develop. Since this type of component doesn't include any graphics or associated code, you may find that development time of the core logic is greatly reduced.

Creating the Component Shell

This section shows you how to create the shell for the very small component that we'll build for this example. With this in mind, all we really need is the ATL shell and a single component as the basis for this example. The procedure in this section is a very simplified version of the procedure in Chapter 13, where we created the first ATL-based component in the book. Make sure you're familiar with the basics of creating a component using ATL before you create the example in this section.

Try This

1. Start Visual C++, if you haven't done so already.
2. Use the File | New command to display the New dialog box. Click the Projects tab.
3. Highlight the ATL COM AppWizard icon.
4. Type a project name in the Project name field, such as **AddIt** in the example.
5. Click OK. You'll see the ATL COM AppWizard - step 1 of 1 dialog box.
6. Choose the Executable (EXE) option. Remember, we're creating an out-of-process server to execute on another machine, so we need to use the EXE option this time. DLLs are used for in-process servers that execute within the same process as the client.

7. Click Finish. You'll see a New Project Information dialog box like the one shown in Figure 14-7.

Figure 14-7: The New Project Information dialog box shows that this is a DLL-based ATL component.

8. Click OK. Visual C++ creates the program shell for you. Remember that this shell is essentially blank; there aren't any objects included in it. The next step is to add an object.
9. Right-click the AddIt Classes folder in the ClassView tab of the Workspace window and select New ATL Object from the context menu. You'll see an ATL Object Wizard dialog box.
10. Select the Objects option in the Category field. You'll see a list of simple objects like the ones shown in Figure 14-8. These objects share one thing in common for the most part — they don't provide a user interface because they're designed to work in the background.
11. Highlight the Simple Object icon, then click Next. You'll see an ATL Object Wizard Properties dialog. This is the same dialog box that we've used before to create objects.
12. Type **AddNumbers** in the Short Name field. Visual C++ automatically fills in the rest of the fields for you.
13. Click OK. Visual C++ creates the new object class for you. At this point, the shell is essentially completed. You do, however, need to add a single method that will allow the application program to add two numbers together.

Figure 14-8: These objects aren't designed to provide a user interface unless you add the required support.

14. Right-click the IAddNumbers interface icon in the ClassView tab of the Workspace window and choose Add Method. You'll see the Add Method to Interface dialog box.
15. Type **DoAdd** in the Method Name field and **[in] int Input1, [in] int Input2, [out] int *Output** in the Parameters field. Notice the use of the [in] and [out] attributes for the various parameters. Any [out] parameter needs to be a pointer instead of an actual value.
16. Click OK. Visual C++ adds the new method to the object.

Adding Some Code

The code for this example is very simple. As previously mentioned, the only purpose it serves is to add two numbers together. However, this example demonstrates in very simple terms how an out-of-process server functionality works whether it is or isn't used remotely. Listing 14-2 contains the component source.

Listing 14-2: **DoAdd() Method Source Code**

```
STDMETHODIMP CAddNumbers::DoAdd(int Input1, int Input2, int *Output)
{
    // Add the two numbers and return them in Output.
    int    iValue = Input1 + Input2;
    *Output = iValue;

    // Tell the client that everything went OK.
    return S_OK;
}
```

Testing the Result Locally

At this point, you can compile both the DCOMTest application and the AddIt component for a local test. When you start the application, you see a simple dialog box like the one shown in Figure 14-9 asking for two numbers as input. Entering the two numbers and clicking Add It displays the result in the Output Number field of the dialog box.

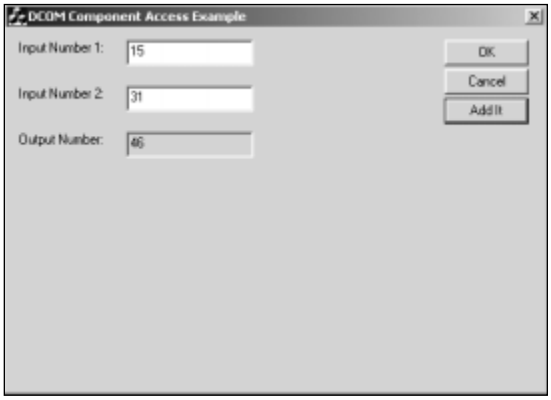

Figure 14-9: The DCOMTest application allows you to test the component locally before running it on another machine.

When you finish testing the component's local operation, copy the component to the machine you want to use as a server and register it there. Normally, you may use RegSvr32 to register either an OCX or DLL version of a component. Out-of-process servers are EXE files and come with their own built-in registration feature. All you need to do is type **AddIt/RegServer** at the command line to register the Add it component. Likewise, typing **AddIt/UnregServer** at the command line unregisters the component on the server (or the client). You can find the code used to register and unregister the component in the AddIt.CPP file.

Using the DCOM Configuration Tool

The DCOM Configuration Tool is a powerful utility that allows you to work with the components registered on a client machine. You can set the component to execute remotely or locally. In addition, this utility allows you to modify the security setup for the component.

Needless to say, this utility also allows you to modify general DCOM operation on a client machine. For example, even though UDP/IP is the normal protocol used to create a connection between the client and the server, you can set it to any of the protocols that both client and server support. This means that you can create a secure connection between client and server by using an encrypted protocol instead of a plain text protocol like UDP/IP.

This section of the chapter begins by showing you how the DCOM Configuration Tool works and how you can use it to redirect execution of the component from the local machine to a server. In this case, let's change the point of execution for the AddIt component; the methods work with any component. Let's also look at some essentials such as security and the use of protocols. Getting the right setup is important if you want your DCOM application to work as anticipated.

Setting Up the General DCOM Environment

Microsoft assumes certain defaults when setting DCOM up on your machine during the Windows installation process. Normally, these defaults work just fine, but you may find that you either need to change the features supported by the DCOM or the protocols used to communicate between client and server.

Obviously, the first thing you need to do before you can reconfigure anything is start the DCOM Configuration Tool. Just use the Run command on the Start menu to display the Run dialog box, then type DCOMCnfg in the Open field and click OK. You'll see the initial Distributed COM Configuration Properties dialog box shown in Figure 14-10. The Applications tab of this dialog box displays all of the out-of-process servers that are installed on the client machine. The out-of-process server must be registered on the client, even if you intend to run it from a remote server. This dialog box reflects only the out-of-process servers that are registered locally.

Figure 14-10: The Applications tab of the Distributed COM Configuration Properties dialog box shows a list of locally registered, out-of-process servers.

The Active Directory support—provided by Windows 2000—obviates the need to register a component locally for use on a remote server. The client workstation downloads global registry entries from the domain server during login. These entries can include remote component registry settings that will allow a client to access the component on a remote server. The advantage to this method is that the administrator only needs to make a single entry on the domain server for DCOM components.

The Default Properties tab shown in Figure 14-11 allows you to change the general characteristics of DCOM on your machine. By default, Microsoft enables DCOM support on a client. This primary level of support will allow you to create trusted connections with servers on a local network. A second checkbox, Enable COM Internet Services on this computer, will allow you to enable COM communications over the Internet.

The Default Authentication Level listbox controls how communication security is handled between the client and server. The default setting is Connect, which means that security is handled only once per connection. I'll talk more about how authentication level is used in the "Setting the Authentication Level" section of this chapter.

The Default Impersonation Level listbox controls what the server can do on the user's behalf when it comes to identification. For example, this setting affects what happens if the server has to create another component to satisfy a user request. How is that component created? Is the component created in such a way that it looks like the user has created it directly? The matter of impersonation level affects what kinds of things the server can do for the client and how it does them. The default setting for this field is Impersonate. The following list defines the various levels of impersonation that DCOM supports.

Anonymous This is the least secure impersonation level. The server performs processing tasks for the client without knowing the identify of the client application. Unfortunately, this setting limits the server to performing tasks that an anonymous user can perform—which means that many resources are out of reach.

Delegate The most secure impersonation level. The server is allowed to perform all tasks as if it were the client. This includes requesting services and resources on other servers. Windows NT Sever authentication service doesn't support this level of impersonation.

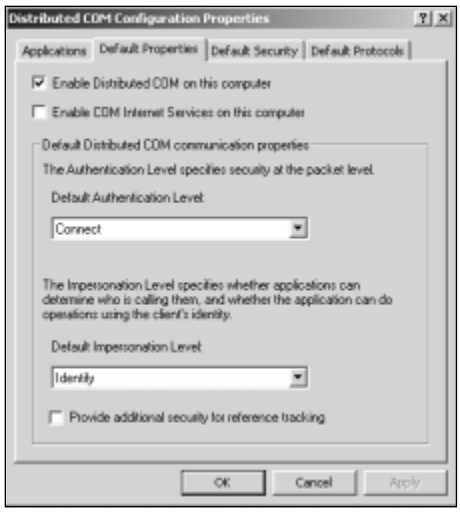
Distributed COM Configuration Properties
Applications
Default Properties
Default Security
Default Protocols
Enable Distributed COM on this computer
Enable COM Internet Services on this computer
Default Distributed COM communication properties
The Authentication Level specifies security at the packet level.
Default Authentication Level:
Connect
The Impersonation Level specifies whether applications can determine who is calling them, and whether the application can do operations using the client's identity.
Default Impersonation Level:
Identity
Provide additional security for reference tracking
OK
Cancel
Apply

Figure 14-11: The Default Properties tab includes settings for the kinds of communication that can take place and the authentication level used for this communication.

Identity Allows the server to identify the client application. What this means is that the client gains full access to the resources and services available on the server as long as it makes the call directly. The server isn't allowed to impersonate the client, even for local service and resource requests.

Impersonate This is the highest level of impersonation level that the Windows NT Server authentication service supports natively. It allows the server to impersonate the client locally, which means that any services or resources that the server has to request to satisfy a client request are done in the client's name. The Impersonation level allows client impersonation only on the local server; a server can impersonate the client on other servers.

The Provide additional security for reference tracking checkbox allows the server to track connected client application requests. Checking this option uses additional server resources — both memory and processor cycles. However, it does enhance server security because a client can't artificially set a server process' reference count to 0, which can kill the process prematurely. The effect of killing a process prematurely is that other clients likely will crash when they attempt to free reference pointers to objects that no longer exist.

The last set of general settings for DCOM appears on the Default Protocols tab of the Distributed COM Configuration Properties dialog box shown in Figure 14-12. This tab shows the various protocols that DCOM attempts to use in creating a connection between a client and server. In addition, the protocols get tried in the order in which they appear in the list. Figure 14-12 shows the default list of protocols that DCOM attempts to use and the default order in which it tries them. Notice that UDP/IP is at the top of the list since this is normally the first choice for DCOM communications, at least on a LAN.

Figure 14-12: The Default Protocols tab allows you to configure the protocols that DCOM attempts to use to create a connection.

There are various controls on the Default Protocols tab that allow you to modify the protocols that DCOM uses. Clicking Add displays the Select DCOM protocol and endpoint dialog box shown in Figure 14-13. This dialog box contains a list of all the potential protocols and endpoints that DCOM supports, not necessarily the ones installed on the client machine. In other words, you have to use some care in choosing a new protocol and endpoint. Although nothing terrible happens if you choose a protocol and endpoint the client can't support, DCOM wastes time trying to make it work, which only delays the process of creating a connection.

Select DCOM protocol and endpoint
Select the DCOM network protocol sequence that you want to add. Enter the endpoint details and click OK when you have finished.
Protocol Sequence:
Connection-oriented SPX
OK
Cancel

Figure 14-13: The Select DCOM protocol and endpoint dialog box allows you to add a new protocol and endpoint to DCOM's support list.

Some protocol/endpoint combinations also allow you to make some configuration choices. For example, highlighting the Connection-oriented TCP/IP option, then clicking Properties, displays a Properties for COM Internet Services dialog box like the one shown in Figure 14-14. This dialog box allows you to choose port ranges for various types of TCP/IP communication. You can choose one set of port ranges for intranet use only and another set of ranges for Internet use only. Security is the main reason to use this configuration dialog box—you need to set specific port ranges when using packet filtering with most firewall products.

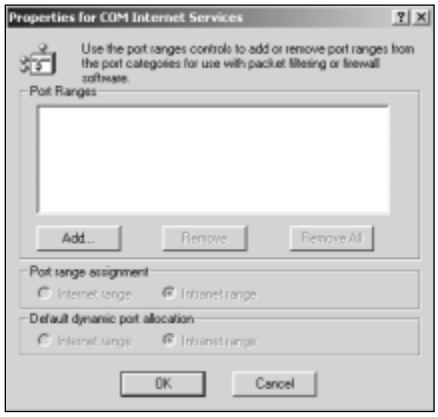

Figure 14-14: Some protocol/endpoint combinations allow further configuration using dialog boxes such as this one.

The order in which protocols get tried can make a big difference in some environments. For example, you may not want to use a connectionless protocol if there's a connection-oriented alternative available. To change the order in which protocols are tried, highlight the protocol, then click Move Up or Move Down as needed to change its order in the list. DCOM attempts to use the protocols in turn starting from the top of the list and going down.

Redirecting a Component

The whole purpose of using the DCOM Configuration Tool, at least from a programming perspective, is to allow an application on the client machine to access a component that you've placed on a server. Distributed processing, coupled with multithreading programming techniques, allows two machines to work in tandem on a single problem. In sum, using DCOM should result in higher system throughput, efficiency, and security, while reducing overall network bandwidth.

This section shows you how to redirect execution of the AddIt component that we created previously from the client machine (most likely your development machine) to a development server. Obviously, some of the particulars of my setup vary from your setup. For example, your development server has a different name from mine. However, except for the names, this procedure will allow you to redirect the execution of any component from the local machine to a server.

Try This

1. Start the DCOM Configuration Tool, if you haven't already done so.
2. Highlight the component you want to redirect on the Applications tab of the Distributed COM Configuration Properties dialog box. For the purposes of this example, let's use the AddIt component. (If you don't see the AddIt component in the list, make sure you registered the component on the test machine using the information in the "Testing the Result Locally" section of this chapter.)
3. Click Properties. You'll see a <Component Name> Properties dialog box like the one shown for the AddIt component in Figure 14-15. Notice that the General tab shown in the figure contains an Authentication Level listbox. I'll talk about this listbox in the "Setting the Authentication Level" section of this chapter.
4. Choose the Location tab. You should see a display similar to the one shown in Figure 14-16. This is where you choose where the component will run. The default setting runs the component on the local machine. There's also an option to run the component wherever the client's data is located. You may find this option handy for database applications where a single client may need to access the same type of data on several different servers. Keeping the component on the server reduces network traffic because only query result information is sent back to the client. The option we're interested in for the AddIt component runs the component on a server, rather than locally.
5. Uncheck the Run application on this computer option and check the Run application on the following computer option. You can use the Browse button to find the server on the network. At this point, you can click OK to make the changes permanent. However, in many cases, you need to configure additional settings to gain access to the server.

Figure 14-15: The initial AddIt Properties dialog box shows general component information, including authentication level.

If you choose more than one location to execute a component, DCOM looks for the first viable execution option and executes the component there. The normal search order is to look at the local machine, then a server that you specify, and finally the location where the client's data is located. If you don't want the client to attempt to execute the component locally, make sure you uncheck this option.

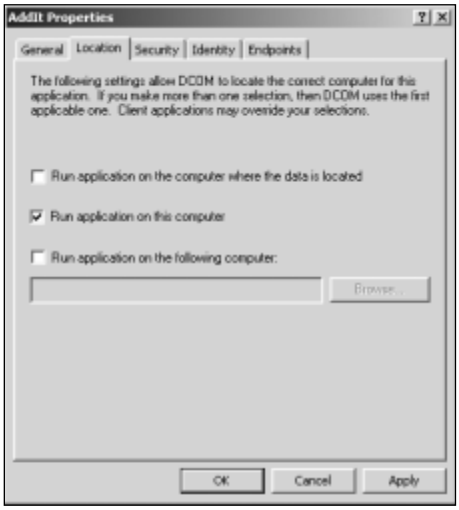
AddIt Properties
General
Location
Security
Identity
Endpoints
The following settings allow DCOM to locate the correct computer for this application. If you make more than one selection, then DCOM uses the first applicable one. Client applications may override your selections.
Run application on the computer where the data is located
Run application on this computer
Run application on the following computer:
Browse...
OK
Cancel
Apply

Figure 14-16: The Location tab allows you to choose one or more places to execute the component.

6. Choose the Identity tab. You'll see a display similar to the one shown in Figure 14-17. This is where you define the identity of the user that executes the component on the server. The default option of the launching user works just fine in most cases (the identity you use to log into the server, which is normally the same as your name and password for the rest of the network). The interactive user option uses the name and password of the person that logged into the workstation and started the client application. This user option allows you to enter the name and password of a user that has the correct level of access to the server for executing the component. Finally, if you want to execute a service remotely (and not a component as we have here), you can choose to use the built-in system account for the server in question.

7. Select one of the identity options. As previously mentioned, the default option of The launching user works just fine in most cases.

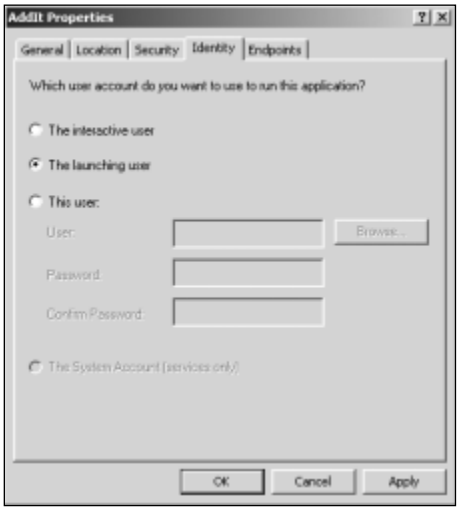

Figure 14-17: The Identity tab allows you to choose which user logs into the server to request component execution.

8. Choose the Endpoints tab. You'll see a dialog box similar to the one shown in Figure 14-18. Normally, DCOM simply uses the protocols that were defined on the Default Protocols tab of the Distributed COM Configuration Properties dialog box. You can, however, choose to add additional protocols/endpoints, add a specific endpoint, or even disable a protocol/endpoint.

9. Click Add to see a Select DCOM protocol and endpoint dialog box such as the one shown in Figure 14-19. You can see the same list of protocols here that you saw on the Default Protocols tab. However, in this case, there are several additional options for configuring those protocols/endpoints. The Disable protocol sequence options allows you to disallow a protocol/endpoint, even if DCOM normally uses it to create a connection. You can also use a static endpoint or the range of port addresses defined for Internet or intranet use. Finally, you can choose the Use default endpoints option, which allows DCOM to use the normal endpoints for the specified protocol. Selecting this last option allows you to change the order in which DCOM chooses protocols for making a connection. DCOM always uses the protocols/endpoints that you define for the component first, then moves on to the general DCOM choices.

Figure 14-18: The Endpoints tab allows you to change the protocol/endpoint that DCOM uses to create a connection for your component.

10. Choose a Protocol Sequence option, then add an Endpoint Assignment if necessary to control the protocol/endpoint configuration. Repeat this step for each custom protocol/endpoint required by your application.

11. Click OK to make the component choices permanent.

12. Click OK to close the DCOM Configuration Tool.

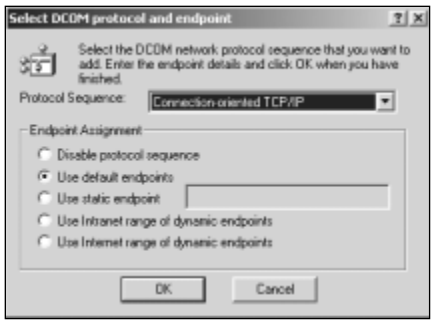
Select DCOM protocol and endpoint
Select the DCOM network protocol sequence that you want to add. Enter the endpoint details and click OK when you have finished.
Protocol Sequence:
Connection-oriented TCP/IP
Endpoint Assignment
Disable protocol sequence
Use default endpoints
Use static endpoint
Use Intranet range of dynamic endpoints
Use Internet range of dynamic endpoints
OK
Cancel

Figure 14-19: You don't have to use the default DCOM protocol/endpoint selections; this dialog box allows you to choose custom settings.

This procedure doesn't address security. I'll cover security issues in the next section of the chapter. For the most part, you don't need to modify the security settings for a component in a development environment. You do need to check with the network administrator for security settings in a production environment.

Creating a Secure Environment

Security is a major concern for any distributed application. The reason is simple: Making the application available over a network means that more than one person can and will access it. The more open your network environment and the more critical the data being transferred, the greater your need for robust security. You need to be sure that the right person accesses the application, while keeping others out. This is especially important in today's Internet-oriented environment where a company may want to allow remote access of critical company information using a standard Internet connection.

In previous sections, we saw some basic security measures you can take when configuring DCOM. For example, you can change the Default Authentication Level to ensure that each communication between client and server is authorized. (This section provides more detailed information on your authentication options.) The default authentication level ensures authentication only when the client connects initially — normally to create the object in the first place or add a reference to an already existing object.

Windows NT uses *Access Control Lists (ACLs)* to control access to most objects on the server. The DCOM Configuration Tool utility can use the security measures provided by Windows NT to secure remote component communications as well. You can actually build an ACL for each component that DCOM will allow users to access remotely. Using an ACL means that only certain people with the right credentials can gain access to the component. In other words, you secure the connection to the server itself and the component on the server, making it unlikely that anyone who isn't authorized can gain access to the component itself.

This section of the chapter shows you how to use the DCOMCNFG utility to secure the components on your server. Using this utility greatly reduces the external threats to your network.

Securing the server and the components that reside on it still doesn't secure the data that gets transferred between client and server. Make sure that you always use some type of secure communication protocol to transfer data from one point to another—especially when working on a public network. Check the various options in the "Setting the Authentication Level" section of this chapter for details on how you can use built-in DCOM settings to make your data more secure. (A good encryption product only enhances the security you can get by default.)

Using the General DCOM Security Options

This section focuses on making sure that the DCOM environment as a whole is secure. You can find the general DCOM security settings on the Default Security tab of the Distributed COM Configuration Properties dialog box as shown in Figure 14-20. Notice that there are three main security areas that this tab covers: access, launch, and configuration. The access area determines who can connect to your machine for the purpose of using DCOM in the first place. The launch area determines who can run applications remotely on your machine. Finally, the configuration options determine who has permission to configure the DCOM settings on your machine. The following list tells you about these areas in more detail.

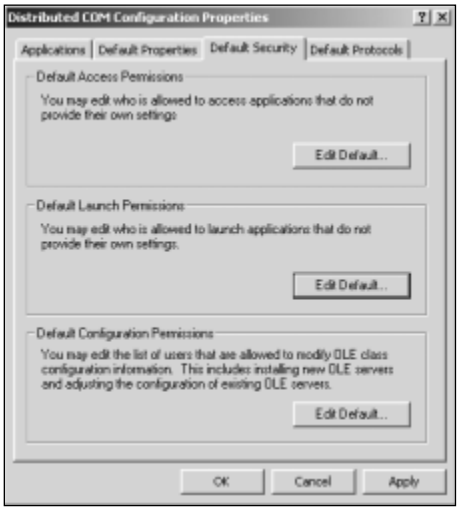
Distributed COM Configuration Properties
Applications
Default Properties
Default Security
Default Protocols
Default Access Permissions
You may edit who is allowed to access applications that do not provide their own settings
Edit Default...
Default Launch Permissions
You may edit who is allowed to launch applications that do not provide their own settings.
Edit Default...
Default Configuration Permissions
You may edit the list of users that are allowed to modify OLE class configuration information. This includes installing new OLE servers and adjusting the configuration of existing OLE servers.
Edit Default...
OK
Cancel
Apply

Figure 14-20: There are three areas of security covered by the options on the Default Security tab.

None of the settings in this section override application-specific settings. Windows 2000 will use the applications-specific settings first.

Default Access Permissions	This option defaults to using the same access-level permissions provided by the system as a whole. In other words, if people have permission to access your hard drive in some way, they usually have permission to access the applications they find there as
Default Launch Permissions	There are three groups that are normally allowed to launch DCOM applications by default: Interactive Users, Administrators, and System. You must specifically assign a user to the Interactive Users or Administrators groups before they have launch permission. Only system-level services are allowed to use the System account. This particular dialog box also allows you to add specific groups that aren't allowed to launch an application. However, you need to remember the rules for using an ACL when adding deny entries. The ACL always goes from the top of the list to the bottom and acts on the first entry it sees. So if a user is part of two groups, one that's allowed to launch DCOM applications and another that isn't, you need to put the deny group first. Otherwise, the user is still allowed to launch DCOM applications based on the group entry that's allowed to launch applications.
Default Configuration Permissions	Anyone who can work with the configuration of DCOM on your machine at all is in this list. Normally, the System account and the Administrators group are allowed full control over the configuration. Everyone else is only authorized to read the configuration. You have to allow read permission to anyone who launches or accesses applications because otherwise DCOM doesn't know how to interact with our machine. There's also a Special Access setting that allows you to customize settings, but this is a difficult entry to set up and normally you won't need to use it.

All three of these settings work the same way, so I'll discuss them as a single entity, even though each area affects a different area of DCOM communication. Clicking Edit Default displays a Registry Value Permissions dialog box like the one shown in Figure 14-21 for launch permissions. This dialog box allows you to add and remove users from the access list.

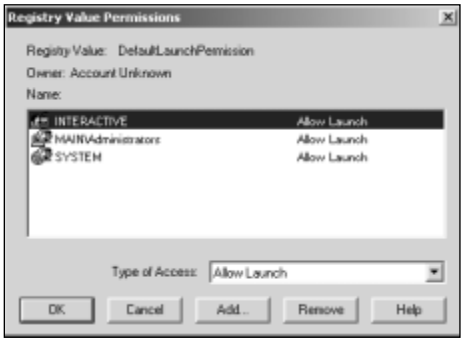

Figure 14-21: The Registry Value Permissions dialog box shows access permissions in addition to those normally set for the machine as a whole.

Click Add to see an Add Users and Groups dialog box like the one shown in Figure 14-22. This particular dialog box shows the list of users and groups for the domain controller; you can also choose from a list for the local machine. There are advantages to using each list. Using the local machine's list allows you to individually control the security settings on a machine-by-machine basis. On the other hand, using the domain list means that you only need to change the settings on one machine to add a new user to a group that affects all machines running DCOM applications.

Working with Component-level Security

Component level security works much like the general security that I just talked about. Figure 14-23 shows a typical example of a component-level Security tab (in this case, for the AddIt component). Notice that you can choose between general security and component-level security for all three groups. If you choose the default security option, then DCOM uses the options set on the Default Security tab of the Distributed COM Configuration Properties dialog box. Otherwise, you need to click the individual Edit button to set the security you want for that area of component level security. Since the actual process of setting component-level security is the same as default security, I won't cover the various options in this section (refer to the previous section).

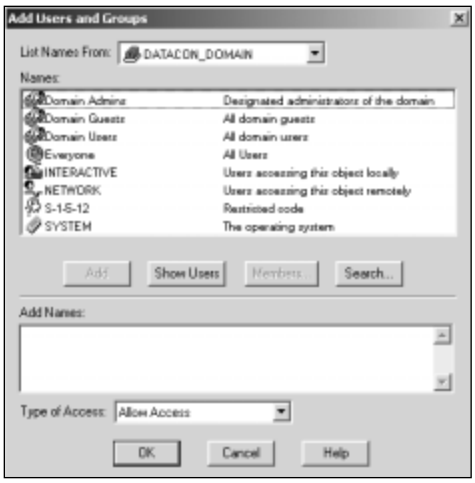

Figure 14-22: DCOM allows you to choose users and groups from the local machine or from the domain controller.

Setting the Authentication Level

There are two ways to set the authentication level for your component: at the DCOM or individual component level. Both authentication level settings determine the minimum-security requirements for the client and server to gain access to each other's resources. In other words, this setting determines how the client and server exchange security information during a session.

You can find the Default Authentication Level setting on the Default Properties tab of the main Distributed COM Configuration Properties dialog box. As previously mentioned, the default general setting is Connect. The component specific setting appears on the General tab of the <Component Name> Properties dialog box. The default setting, in this case, is Default. These two settings represent a minimalist approach to DCOM security. There are other settings that may provide you with better security. Table 14-5 describes each of the Default Authentication Level settings.

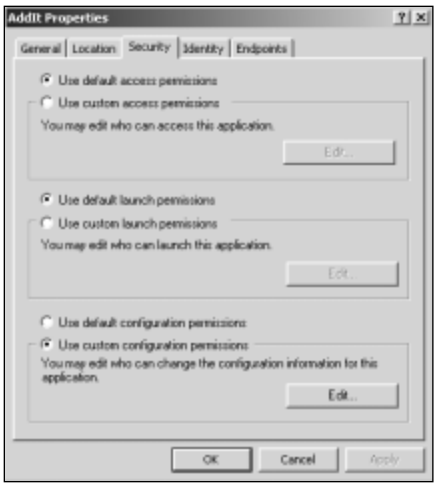

Figure 14-23: The major difference between the component-specific Security tab and the Default Security tab is default/custom option buttons.

Table 14-5
DCOM Default Authentication Level Settings

Setting	*Description*	*When to Use*
None	The client and server don't authenticate each other at all.	You must use this setting when creating anonymous connections. However, this setting also comes in handy working in a small workgroup network situation where physical security is high and there aren't any connections to the outside world. This setting reduces overall network traffic to the lowest possible level, but also represents the greatest security risk.
Call	Authentication occurs for every call while a connection is maintained.	This is a moderately high security level that ensures the client and server verify each other's identity for each method call in the application, which may involve several packets. You may notice a slight increase in network traffic when using this level of authentication, and there's no guarantee of complete security from third-party intrusion.

Continued

Table 14-5 (*continued*)

Setting	*Description*	*When to Use*
Connect	One-time authentication takes place during object creation.	The client and server verify each other's identity during this initial request. There's a good chance a third party can break network security when using this mode since the client and server don't have any way to verify either packets or requests. The advantage to this method is that initial authentication places limits on what the user can do, and network traffic is kept to a minimum.
Default	Standard setting used by DCOM.	At the individual component level, this setting means that the component uses the general DCOM setting. At the DCOM level, this setting means that DCOM as a whole uses whatever security the authentication method uses in general. For example, the default Windows NT Security Services uses Connect level authentication. The results of using this setting vary according to the authentication method used.
Packet	A minimal level of secure . protection that encrypts the user's identity	This is the first of three levels of truly secure DCOM communication settings. The sender's identity is encrypted and packaged with the packet. This means that the receiver can verify the authenticity of the sender with every packet and greatly reduce the probability of third-party intrusion. However, this method also bloats the size of the packet and can greatly increase DCOM-related network traffic. This setting represents the most reasonable level of protection for a network that allows outside access.
Packet Integrity	The sender's identity and a packet signature .are encrypted as part of the packet	Using these two forms of authentication ensures that the sender is authorized and that the packet hasn't been modified in any way. However, this method doesn't ensure that a third party hasn't read the packet, and it does increase network traffic over Packet authentication. You only need to use this setting when the integrity of the data is absolutely essential, but you don't care who reads the packet.

Setting	*Description*	*When to Use*
Packet Privacy	Paranoid security for extreme situations.	In most cases, this is the paranoid level of security. Not only does the packet contain the sender's identity and a packet signature, but the packet itself is also encrypted to ensure that a third party can't read it. This level of authentication greatly increases network traffic and can actually slow communications to a crawl when used on slower traffic media such as a dial-up connection. However, this is the level of authentication you need to ensure safe financial transactions and the transmittal of critical confidential information from one site to another.

Performing a Remote Test of the AddIt Component

At this point, you should have configured the AddIt component for execution on a remote development server. You don't need to recompile the application or component, or do anything else special. Changing the registry using the DCOM Configuration Tool is all you really need to do.

There's an easy way to verify that the component is executing on the remote server. Right-click the Task Bar on the remote server (the bar normally found at the bottom of the Windows display) and choose Task Manager from the context menu. You'll see a Windows Task Manager dialog box. Click the Processes tab to see a list of processes executing on the machine. Click the Image Name column header to view the process names in alphabetical order (this helps you find a process faster). Look through the list of processes; you shouldn't see AddIt.EXE anywhere in the list. Now, run the DCOM application, type two numbers in the input fields, then click Add It. You'll see the AddIt.EXE component added to the process list as shown in Figure 14-24. After a few seconds, the component disappears from the list. This is your assurance that the component has executed on the machine that you indicated.

Check out the same display on the client machine. What you see is that DCOMTest.EXE appears on the list, but that AddIt.EXE doesn't. This may seem like an overly simplified way to see DCOM in action, but it does show you how even complex setups work in the real world.

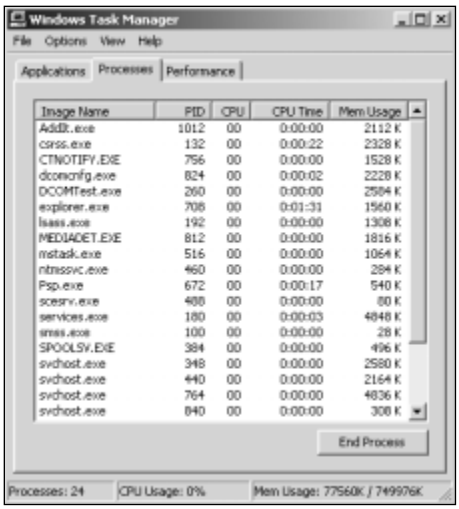
Windows Task Manager
File Options View Help
Applications Processes Performance
Image Name PID CPU CPU Time Mem Usage
AddIt.exe 1012 00 0:00:00 2112 K
csrss.exe 132 00 0:00:22 2328 K
CTNOTIFY.EXE 756 00 0:00:00 1528 K
dcomcnfg.exe 824 00 0:00:02 2228 K
DCOMTest.exe 260 00 0:00:00 2584 K
explorer.exe 708 00 0:01:31 1560 K
lsass.exe 192 00 0:00:00 1308 K
MEDIADET.EXE 812 00 0:00:00 1816 K
mstask.exe 516 00 0:00:00 1064 K
ntvssvc.exe 460 00 0:00:00 284 K
Psp.exe 672 00 0:00:17 540 K
scesrv.exe 488 00 0:00:00 80 K
services.exe 180 00 0:00:03 4848 K
smss.exe 100 00 0:00:00 28 K
SPOOLSV.EXE 384 00 0:00:00 496 K
svchost.exe 348 00 0:00:00 2580 K
svchost.exe 440 00 0:00:00 2164 K
svchost.exe 764 00 0:00:00 4836 K
svchost.exe 840 00 0:00:00 308 K
End Process
Processes: 24 CPU Usage: 0% Mem Usage: 77560K / 749976K

Figure 14-24: Using the Processes tab of the Windows Task Manager dialog box allows you to monitor the execution of components.

Connecting to a Specific Object Instance

There are times when you may want to connect to an existing instance of an object rather than create a new instance of the object. For example, you may need to access a running component from within a thread of an application or as a connecting link from another application.

This section of the chapter looks at the problems of connecting to a running object instance. Obviously, connecting to an existing object not only entails problems with the object itself such as managing state information, but also problems with managing the lifetime of that object (you don't want the object to self-destruct if there are references to it). The following sections help you understand both of these issues.

Object Lifetime Management

When an application programmer creates a standalone application designed to operate on a single machine, there's no doubt as to the lifetime of the objects that application creates. The objects are no longer needed when the user or the system closes the application. The reason is simple: The application using the objects determines their lifetime because they're part of the application itself. Of course, this scenario assumes that the developer has created everything within the application, and nothing within the application is accessible to the outside world.

Things change a little when an application is composed of components that are created by other programmers and the application isn't certain when the object instances are no longer needed. In this case, COM uses reference counting. Every time an object gains a new connection, its reference count rises. Likewise, when an application no longer needs the object, COM reduces the reference count by one. When the reference count reaches zero, the object is destroyed. This is the typical scenario for an in-process server (normally a DLL of some type).

The problem becomes even more difficult to manage when an application and its associated components execute in more than one process. There are times when this happens on the local machine. For example, if you create a compound document, the embedded document requires the services of a server that exists outside of the current process for display and editing needs. Every out-of-process server (normally an EXE) falls into this category. In this case, reference counts alone won't allow proper object lifetime management since there are connections from outside of the current process.

DCOM applications, which execute on more than one machine, always fall into the out-of-process server category since they are no longer in a single process environment. Although DCOM is actually COM with a longer wire, it treats every component as an out-of-process server even though many of the components appear to the client as an in-process server. In other words, DCOM does add complexity to the object lifetime management problems that occur when using components.

To give you a better idea of why a reference alone can't do the job, consider the following scenario. Begin by thinking about what happens when a user embeds a Corel Draw drawing into a Word document. Every time the user opens the Word document, Word creates an instance of the Corel Draw out-of-process server to handle the display of the Corel Draw drawing embedded in the Word document. When the user closes the document, the reference count for the Corel Draw out-of-process server goes to 0 and the object destroys itself.

What happens when the user clicks on the Corel Draw drawing within the Word document, uses Edit | Select All to grab the entire document, copies it, then embeds it into an Excel document? When the user opens the document in Excel, there actually are two references to the Corel Draw out-of-process server. The first is obvious; Excel needs a reference to the Corel Draw out-of-process server to display the figure. The second comes from Word, which acts as the container for the Corel Draw drawing. When the user closes Excel, the first reference to the Corel Draw out-of-process server is released, but since Excel has no direct link to Word, the second reference remains in place.

This is where the concept of strong and weak references comes into play. A strong reference ensures that an object remains open, whereas a weak reference doesn't. In other words, Excel has a strong reference in our example, whereas Word has a weak reference. The use of a weak reference allows Word to act as a container for the Corel Draw document, but isn't enough to keep the document open after Excel is closed by the user. Both the Word and Corel Draw out-of-process servers are released automatically when the user closes Excel, which is how things are supposed to work from the user's perspective. (All this assumes that every programmer out there observes every COM rule, which doesn't happen. This section looks at a perfect scenario where all of the components behave exactly as anticipated.)

What actually happens is that the Corel Draw out-of-process server keeps track of both locks (strong references) and weak references. When the number locks goes to zero, Corel Draw's out-of-process server issues a Close( ) method call. The Close( ) method does everything needed to "encourage" the reference count to go to zero. This includes revoking its running object table (ROT) entry (so that no other clients can attach to the current instantiation of the object), sending an OnClose( ) notification to any attached clients with weak references, closing any internal pseudo-objects, and disconnecting any remote connections. Once the reference count goes to zero, the object is destroyed and the memory is freed.

Working with the Running Object Table

The previous section looks at the requirements for maintaining the lifetime of an object. One part of the discussion involves two links to the Corel Draw out-of-process server: a weak one from Word and a strong one from Excel. This means that somewhere along the way, Word created a reference to the already running Corel Draw object. Obviously, compound documents are just one time when multiple references to the same object are needed, but this is a very good example because it's one that's commonly used.

Of course, the question is how Word obtained the information necessary to create a reference to the Corel Draw out-of-process server. Components that will allow a client to create a connection to a running instance need to register themselves in a special COM table called the *running object table (ROT)*. In short, the ROT contains a reference to the object so clients can attach to it.

There are two levels of registration: strong and weak. The main difference between strong and weak ROT registrations is that a strong registration prevents an object from shutting down until all of the connected clients have also shut down. A weak reference gets shut down automatically once all strong references to the object are cleared. In most cases, you want to use the default registration level of weak.

Objects register themselves in the ROT using the IRunningObjectTable::Register( ) method. The object itself determines whether the registration is strong or weak. The default is to create a weak registration. Adding the ROTFLAGS_REGISTRATIONKEEPSALIVE flag creates a strong registration. In addition, the object can control which clients get to create a reference to the object. The default allows only clients that are part of the original application window to gain access to the object. However, adding the ROTFLAGS_ALLOWANYCLIENT flag allows any client to create a reference to the object. An object removes itself from the ROT using the Revoke( ) method.

A client that wants access to a given object uses the GetObject( ) method, which requires two arguments. The first argument contains the name of the object to look for in the ROT. The second argument is a pointer to a variable that receives the IUknown interface pointer for the object if the call is successful. If a client simply wants to check if an object is available, it can use the IsRunning( ) method.

Microsoft provides a handy utility for viewing the ROT called the ROT Viewer. I discuss this utility later in this chapter.

DCOM and Monikers

DCOM provides an array of methods for identifying the specific object instances that an application needs to use. Although a unique CLSID and (optionally) a server name always identify the object itself, the instantiated object can be identified by a variety of methods such as a URL for a Web page. The process of naming a specific instance of a COM object is called creating a moniker. The moniker is used to identify the object.

The following sections of the chapter visit several issues related to DCOM and its use of monikers. Obviously, the first topic we need to cover is how monikers work in the first place. Part of this operational view of monikers requires the use of the ROT Viewer utility supplied with Visual C++. The second section tells you about this utility and how you can use it to better understand how the objects on your machine are accessed.

How Monikers Work

The monikers supported by DCOM can vary a great deal in part because the moniker itself is an object. The moniker object contains the logic necessary to find the running instance of the object that it identifies. In fact, like so many other areas of COM, there's a standard interface designed to perform this work known as IMoniker.

Before you get the idea that every component you create will appear in the ROT with a moniker attached, you need to know that the moniker refers to a file or other storage reference of some sort. In fact, there are five different types of monikers supported by Windows, each of which has a special purpose. The following list helps you understand the various moniker types (the two most common of which are file and item monikers).

File Moniker — A standard file object. Whether the file appears on the Internet, a WAN, a LAN, or on your local hard drive is immaterial — the moniker binds an object and a file together. The object then registers this combination in the ROT, which allows other applications to interact with that object. You can use the CreateFileMoniker( ) method to create a file moniker. This method requires you to input parameters: the name and path to a file, and a pointer to the variable used to hold the interface pointer.

Item Moniker — Not every object that you need to reference is a file; some objects are contained within files or other objects. An item moniker references an object stored within a container of some type. Item monikers can also be used to refer to pseudo-objects such as a range of cells in a spreadsheet. Use the CreateItemMoniker( ) method to create an item moniker. This function requires three inputs: a delimiter used to separate the object's name from the container's name in the ROT display, a string containing the name of the object, and the address of a variable used to hold the IMoniker interface pointer.

Generic Composite Moniker — A combination of more than one moniker into a single moniker. The monikers don't have to be of the same type, which means you can use different classes of monikers in tandem. Use the CreateGenericComposite( ) method to create a generic composite moniker. This method requires three parameters as input: a pointer to the first moniker, a pointer to the second moniker (which can be another generic composite moniker), and the address of a variable used to hold the resulting IMoniker interface pointer.

Anti-Moniker	This is the inverse of a file, pointer, or item moniker. It's used primarily to create a relative reference to an object. In essence, you create a moniker that provides a relative reference of one moniker when compared to another moniker. (It helps to think of this relationship as a relative file path — the relative file path is a reference to a particular file using the current position in the directory structure as a starting point.) Use the CreateAntiMoniker( ) method to create an anti-moniker. This method requires only one input parameter: the address of a variable used to hold the resulting IMoniker interface pointer.
Pointer Moniker	A reference to an object in memory. This is the only moniker type that can't be saved to permanent storage. Use the CreatePointerMoniker( ) method to create a pointer moniker. This method requires two inputs: a pointer to the interface that you want to create a moniker for, and the address of a variable used to hold the IMoniker interface pointer.

Using the Running Object Table (ROT) Viewer

Now that you have a better idea of how monikers and DCOM work together, it's time to see the ROT in action. Microsoft provides the running object table (ROT) Viewer utility for the purpose of seeing what objects are registered in the ROT. Not only is this a great diagnostic tool for your own applications, but it provides a great way to find out more about how other applications work.

You can find the ROT Viewer listed with the other Visual Studio/Visual C++ tools installed on your machine. Figure 14-25 shows a typical example of a ROT Viewer window. In this case, we're looking at the compound document created in Excel used earlier for explanation purposes. (The Excel and Word documents are on the CD-ROM, along with the embedded image file.)

The top half of the ROT Viewer window contains a list of 32-bit applications that have registered an object with the ROT. An application can register more than one object. In fact, this example shows that Excel has registered two objects: one is a file moniker and the other is an item moniker.

The bottom half of the ROT Viewer window contains information about the object currently selected in the top half. Figure 14-25 shows a file moniker. Notice that a file moniker entry includes the path and file name for the file. There's also status information that tells the last time the file was modified and whether the application is running. In addition, the ROT Viewer tells you what kind of object you're seeing.

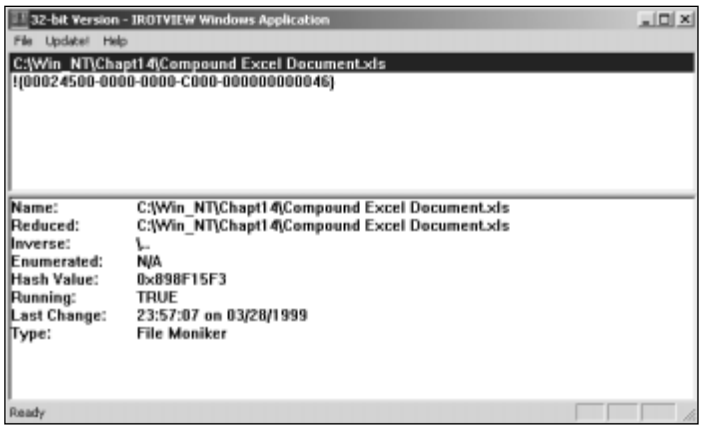

Figure 14-25: This is a typical example of a file moniker.

Select the second object in the top half of this display and you'll see the item moniker display shown in Figure 14-26. Notice that this display contains some of the same information as the file moniker. We know that this is an item moniker, when it was last modified, and that the application it belongs to is running. Notice that the item moniker uses a delimiter of an exclamation point (!). On the left side of the item moniker Name field is the embedded object name (blank in this case); on the left side is the container name (a class identifier in this case). Look up the class identifier in the Registry Editor to find that it refers to a Microsoft Excel 97 Application as shown in Figure 14-27.

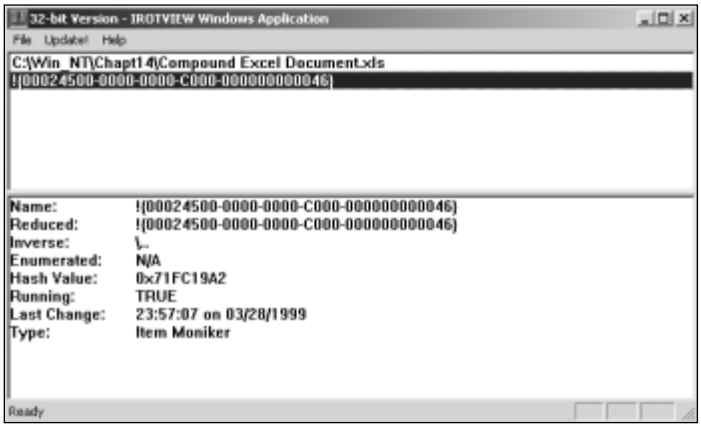

Figure 14-26: Item monikers normally have an object name, a delimiter, and a container name in the moniker Name field.

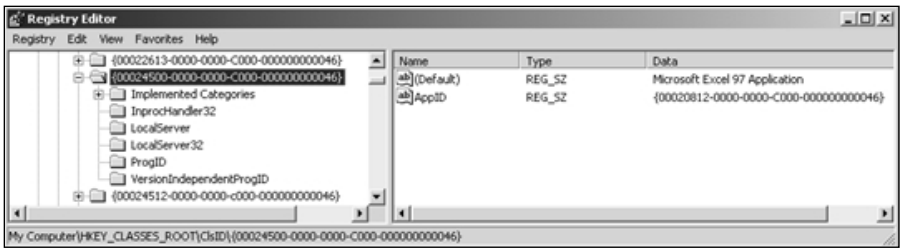

Figure 14-27: It pays to look up the class identifier for a ROT entry in the registry to verify the container name.

Using the Remote Automation Connection Manager

The Remote Automation Connection Manager was originally included with Visual Basic to make creating remote connections a lot easier. You can find it distributed with the Enterprise Edition of most Microsoft programming products (the Professional Edition lacks this tool). Though we won't use the Remote Automation Connection Manager for that purpose, it does provide another good way to look at your DCOM connections. Figure 14-28 shows a typical example of the Remote Automation Connection Manager display. Notice that I've already selected the AddIt control.

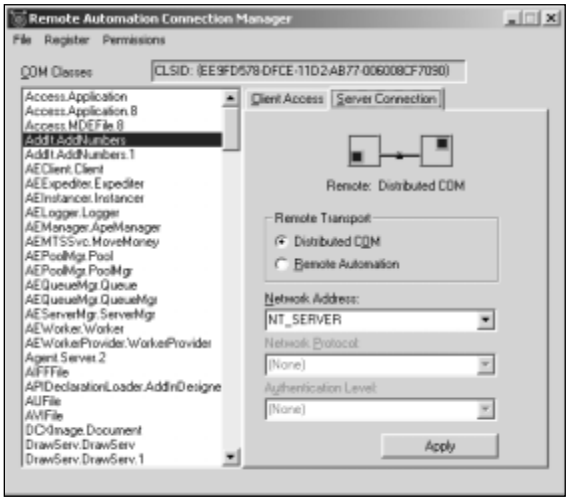

Figure 14-28: The Remote Automation Connection Manager provides another way of looking at your DCOM connections.

Your display may not show the CLSID field shown in Figure 14-28. You can activate this field using the Register | Show CLSID command or by pressing Ctrl+C.

The Remote Automation Connection Manager display is actually a little easier to understand than the DCOM Configuration Tool display that we worked with earlier. Although it isn't nearly as complete in some ways, it's a little more complete in others. For one thing, you get to see all of the registry entries, not just applications. This more complete display allows you to see items that the DCOM Configuration Tool hides — making it possible to find errant registry entries rather than try to dig the information out of your code. For example, an errant registry entry may prevent you from seeing a component that you just created, making it difficult to determine whether the problem lies in your registration code or in the registry entries themselves. If you see the missing component in the Remote Automation Connection Manager, then you know that your registration code is fine and that it's the registry entries themselves that are to blame.

Unfortunately, creating the connection is counterintuitive. There isn't a browse button that allows you to find the server you want to connect to. The Remote Automation Connection Manager won't allow you to change the protocols used for the DCOM connection either — which may actually be a plus for developers who just want to create the connection and don't really care how it gets made.

Fortunately, you can set security at the individual component level, but not for DCOM as a whole. To set individual component security, use the Permissions | Edit ACL command. You'll see the Remote Class Permissions dialog box shown in Figure 14-29. This is where you can select what kinds of access each group can have to a particular object.

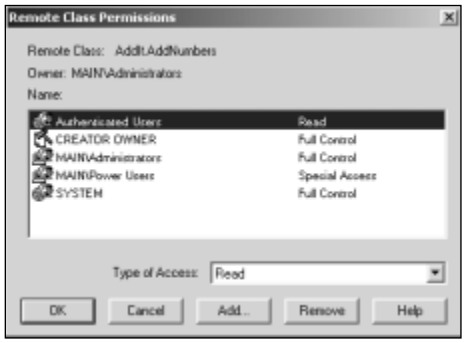

Figure 14-29: The Remote Automation Connection Manager allows you to change the security attached to individual components.

Now, let's say that you want to find out what special access the MAIN\Power Users have in Figure 14-29. All you need to do is double-click their entry in the Remote Class Permission dialog box. You'll see a Special Access dialog box like the one shown in Figure 14-30. This dialog box allows you to control precisely what a particular group can do when accessing or launching a control from a remote computer.

As you can see, the Remote Automation Connection Manager isn't a tool that you use in lieu of the DCOM Configuration Tool — they serve different purposes. It's important to realize that both tools are different. The Remote Automation Connection Manager is actually a better tool for creating quick connections. On the other hand, the DCOM Configuration Tool is better for initial setups and occasional custom setups.

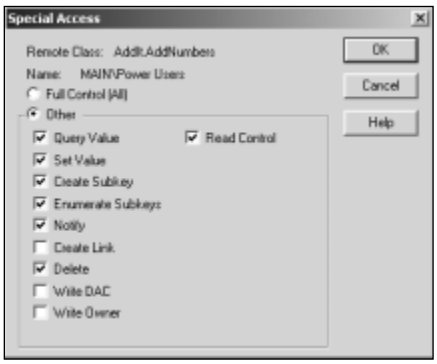

Figure 14-30: The Special Access dialog box allows you to change the type of access a particular group has to a component.

Summary

DCOM provides one way of creating a distributed computing environment. You can use it to allow one machine to borrow resources from another machine on the network, including the components contained on that machine. This chapter talks about the theory and implementation of DCOM. We've seen how DCOM gives the programmer additional flexibility in creating remote connections in a world that's ever more Internet and intranet bound. Here are some additional conclusions you can draw from the contents of this chapter.

- DCOM extends COM so that it can work with remote connections. In short, DCOM is a protocol that manages the remote connection in a manner that's transparent to both the client and the server. All local connections are still managed by COM and its associated technologies.

- The main reasons that you should use DCOM are for fault tolerance, scalability, load balancing, ease of maintenance, flexibility, and connection management.
- There are some situations where you shouldn't use DCOM because of security, performance, or connection availability concerns. Make sure you look at other alternatives before you decide to use DCOM.
- DCOM relies on DCE RPC to get data packets ready for transmission from one machine to another.
- DCOM can rely on a variety of security providers to ensure safe client/server communications. These providers include NT LAN Manager (NTLM, the standard Windows NT security protocol), Kerberos, Distributed Password Authentication (DPA, which is used by CompuServe and MSN), secure channel security services such as Secure Sockets Layer (SSL)/ Private Communication Technology (PCT), and third-party Distributed Computing Environment (DCE) providers.
- There are two standard methods of load balancing for DCOM: static (which relies on registry entries or server references within the client) and dynamic (which relies on a referral component).
- COM+ allows you to load balance components using a combination of a load-balancing router and an application of up to eight servers. The client requests component services from the router, which determines the server used to complete the request.
- There are several names for the DCOM Network Protocol including DCOM Wire Protocol and Object Remote Procedure Call (ORPC).
- The main difference between the DCOM Network Protocol and standard DCE RPC is that DCOM adds the ability to pass object references using an IPID.
- The IRemoteActivation interface is RPC specific. It contains a single method, RemoteActivation( ), that allows DCOM to pass object information between the client and server.
- DCOM relies on the same NDR format that DCE RPC uses to marshal data between the client and server.
- There are five methods that will allow you to access a component from within a client application using DCOM: CoCreateInstanceEx( ), CoGetInstanceFromFile( ), CoGetInstanceFromStorage( ), CoGetClassObject( ), and CoGetClassObjectFromURL( ). The first four rely on a COSERVERINFO data structure to transfer the server information, whereas the fifth relies on the contents of the szCodeURL argument.
- Client applications that use CoCreateInstance( ) to create new objects require two registry entries to enable DCOM access: one for the remote server name and a second for the application identifier.
- The DCOM Configuration Tool allows you to perform two main tasks: redirect component execution from a client machine to a server and set DCOM

security. In addition, you can use this utility to define DCOM features like the protocol used for communication between client and server.

- In most cases, there's both a component level and default level of control for each DCOM setting. The component-level setting always takes precedence.
- Windows NT Server authentication service supports three levels of impersonation: Anonymous, Identity, and Impersonate. A fourth level of impersonation, Delegate, is available for use, but isn't supported by the Windows NT Server authentication service.
- DCOM supports multiple levels of authentication, including None, Call, Connect, Default, Packet, Packet Integrity, and Packet Privacy. Of these, only Packet Privacy fully protects your data from both third-party scrutiny and modification.
- There are two main issues when connecting to an existing object: object lifetime and gaining access to the object IUnknown interface pointer. The object lifetime is automatically maintained through the use of strong and weak references. The ROT allows an object to register itself so clients can create the required connections.
- There are five different kinds of monikers that you can bind to object, then register in the ROT. They include file moniker, item moniker, generic composite moniker, anti-moniker, and pointer moniker.
- The ROT Viewer allows you to see the objects that are registered in the ROT. The two most common object types are file monikers and item monikers.
- The Remote Automation Connection Manager provides another method for working with DCOM connections. In this case, the utility provides the means to create quick connections without wading through a cumbersome interface.

✦ ✦ ✦